



AFRL-RI-RS-TR-2011-003

**JVIEW VISUALIZATION FOR
NEXT GENERATION AIR TRANSPORTATION SYSTEM**

CACI TECHNOLOGIES INCORPORATED

JANUARY 2011

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

STINFO COPY

**AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (<http://www.dtic.mil>).

AFRL-RI-RS-TR-2011-003 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/
PETER A. JEDRYSIK
Work Unit Manager

/s/
JULIE BRICHACEK, Chief
Information Systems Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE*Form Approved*
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Service, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**1. REPORT DATE (DD-MM-YYYY)**

January 2011

2. REPORT TYPE

Final Technical Report

3. DATES COVERED (From - To)

September 2007 – September 2010

4. TITLE AND SUBTITLEJVIEW VISUALIZATION FOR
NEXT GENERATION AIR TRANSPORTATION SYSTEM**5a. CONTRACT NUMBER**

FA8750-07-C-0209

5b. GRANT NUMBER

N/A

5c. PROGRAM ELEMENT NUMBER

35111F

6. AUTHOR(S)Daniel Krisher
Aaron McVay
Patrick Fisher**5d. PROJECT NUMBER**

NASA

5e. TASK NUMBER

NG

5f. WORK UNIT NUMBER

02

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)CACI Technologies, Incorporated
14370 Newbrook Drive
Chantilly VA 20151-2218**8. PERFORMING ORGANIZATION
REPORT NUMBER**

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)Air Force Research Laboratory/RISB
525 Brooks Road
Rome NY 13441-4505**10. SPONSOR/MONITOR'S ACRONYM(S)**

AFRL/RI

**11. SPONSORING/MONITORING
AGENCY REPORT NUMBER**

AFRL-RI-RS-TR-2011-003

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. PA# 88ABW-10-6447

Date Cleared: 10 December 2010

13. SUPPLEMENTARY NOTES**14. ABSTRACT**

This Final Technical Report discusses the accomplishments of an effort to support NASA and FAA goals for visualization of the National Airspace System (NAS) to aid in the analysis of proposed changes in order to increase its capacity to meet future needs. Using JView technology, CACI enhanced an application called the Airspace Concepts Evaluation System (ACES) Viewer they developed for NASA under a previous project. The ACES Viewer is an information visualization tool designed to provide visual representations of the output of the ACES NAS simulation developed at NASA. The application provides mechanisms to load the various types of data used and output by the ACES simulation, process the data, and then display it. The ACES Viewer is not limited to visualization of ACES data, as any data (in one of the supported formats) can be loaded and displayed. The application has been enhanced during this effort to serve more as an aeronautical research tool rather than simply a presentation tool. It has also been enhanced to serve as a platform for several other domain specific visualization tools.

15. SUBJECT TERMS

National Airspace System visualization, Airspace visualization, de-cluttering visualization, overlapping data visualization

16. SECURITY CLASSIFICATION OF:

a. REPORT

U

b. ABSTRACT

U

c. THIS PAGE

U

**17. LIMITATION OF
ABSTRACT**

UU

**18. NUMBER
OF PAGES**

157

19a. NAME OF RESPONSIBLE PERSON

PETER A. JEDRYSIK

19b. TELEPHONE NUMBER (Include area code)

N/A

TABLE OF CONTENTS

TABLE OF CONTENTS.....	i
LIST OF FIGURES	vii
1.0 EXECUTIVE SUMMARY.....	1
2.0 OBJECTIVE	2
3.0 INTRODUCTION	2
4.0 TECHNICAL TASKS	3
4.1 JView	3
4.1.1 Basic Infrastructure.....	3
4.1.1.1 OpenGL Bindings	3
4.1.1.2 High Precision Locale.....	4
4.1.1.3 Geodesic Bounding Box with Source Projection.....	5
4.1.1.4 Stereoscopic Rendering.....	6
4.1.1.5 GLSL Shader Program Abstractions.....	8
4.1.1.6 OpenGL Lighting.....	8
4.1.1.7 Geographic Coordinate Systems	8
4.1.1.8 Rendering Appearance Settings	9
4.1.1.9 Animation Framework	9
4.1.1.10 Vertex Buffer Objects	10
4.1.1.11 Texture API.....	11
4.1.1.12 Picking/Selection	11
4.1.1.13 Bounding Volumes	13
4.1.1.14 Camera System	14
4.1.2 Utilities.....	15
4.1.2.1 Native Library Loader.....	15

4.1.2.2	Camera Navigation	15
4.1.2.3	DTED Interpolation	16
4.1.2.4	View Footprint Calculator	17
4.1.2.5	Movie Capture.....	18
4.1.2.6	Transfer Function Editor.....	18
4.1.3	Rendering Elements	19
4.1.3.1	Callouts	19
4.1.3.2	Labels Element.....	20
4.1.3.3	Instance Rendering.....	24
4.1.3.4	HUD Utilities	25
4.1.3.5	Histogram Grid	25
4.1.3.6	Sphere Segment Scene Element.....	26
4.1.3.7	Pin Renderer.....	27
4.1.4	Model Formats	28
4.1.4.1	Open Flight Models (FLT).....	28
4.1.4.2	Keyhole Markup Language (KML)	29
4.1.4.3	Collada	32
4.1.5	Release Management	37
4.1.5.1	Code Quality	37
4.1.5.2	Continuous Integration.....	38
4.1.5.3	Application Installers	39
4.1.5.4	JView Releases	39
4.1.5.5	JView Training.....	41
4.2	JView World	41
4.2.1	Rendering and API.....	41
4.2.1.1	Node Change Listener API	41

4.2.1.2	Camera Location Based Level of Detail (LoD)	41
4.2.1.3	Iterative Interpolated Geometry Refinement	43
4.2.1.4	Elevation Interpolation.....	43
4.2.1.5	Vertex Buffer Objects	43
4.2.1.6	DTED Level 5	44
4.2.1.7	Mutable World Data Configuration	44
4.2.1.8	Multi-Threaded Texture Fetching.....	44
4.2.2	Imagery	44
4.2.2.1	GeoTIFF.....	44
4.2.2.2	Google Maps Imagery.....	45
4.2.2.3	FalconView Imagery.....	46
4.2.2.4	Layer Manager	46
4.2.2.5	Texture Masking	49
4.2.2.6	Texture Union Optimization	50
4.2.2.7	Pixels Per Degree Threshold Optimization.....	52
4.2.2.8	TextureSet API.....	52
4.2.2.9	RPF Catalog	53
4.2.2.10	Graticule Texture Set	53
4.2.3	World-Specific Scene Elements.....	54
4.2.3.1	World Raster Image Element.....	54
4.2.3.2	Grounded Scene Elements	56
4.2.3.3	World Info Element	56
4.2.4	Profiling and Debugging Tools.....	57
4.2.4.1	World Profiler	57
4.2.4.2	Profiler Histograms	59
4.2.4.3	Texture Passes Histogram.....	60

4.2.4.4	World Automated Test Program	61
4.2.5	Tools	64
4.3	Terra Firma	65
4.3.1	Terrain Profile Visualization.....	65
4.3.2	Geographic Locations Database.....	68
4.3.3	Geographic Search User Interface	70
4.3.4	WebStart	72
4.3.5	Map Server.....	72
4.3.6	Drag and Drop GeoTIFF.....	72
4.3.7	Overview Navigation	73
4.4	ACES Viewer.....	74
4.4.1	Basic Infrastructure.....	74
4.4.1.1	Visualization Modeling.....	74
4.4.1.2	Configuration and Visualization Persistence	74
4.4.1.3	Module and Plug-in Framework	76
4.4.1.4	Dependency Injection	77
4.4.2	Rendering and Visualization.....	78
4.4.2.1	General Enhancements.....	78
4.4.2.2	Display Containers.....	78
4.4.2.3	Renderers	81
4.4.3	Data Access.....	86
4.4.3.1	Memory Consumption and Efficiency Enhancements.....	86
4.4.3.2	CSV Data	87
4.4.3.3	Database (RDBMS) Data.....	88
4.4.3.4	CTAS Data.....	89
4.4.3.5	Weather Polygons	89
4.4.3.6	XML Air Zones.....	89
4.4.4	Data Processing.....	90

4.4.4.1	Units	90
4.4.4.2	Concurrency	90
4.4.4.3	Expressions	92
4.4.4.4	Transforms	95
4.4.5	User Interface.....	97
4.4.5.1	Basic User Interface Components.....	97
4.4.5.2	Docking Frames	98
4.4.5.3	Visualization Controls.....	99
4.4.5.4	Datasets	101
4.4.5.5	Property Sheet	104
4.4.5.6	Visualization Composition Graph.....	107
4.4.5.7	Visualization Composition Toolbox	110
4.4.5.8	Movie Capture.....	111
4.4.5.9	Development and Debugging Tools.....	112
4.4.6	Release Management	113
4.4.6.1	Native Packaging	113
4.4.6.2	Releases.....	114
4.5	Characterization of the UAV Network Environment (CUNE) Viewer	115
4.6	JWeather	122
4.6.1	Background Research on Contemporary Weather Visualization Techniques	122
4.6.2	Access to Real-Time Weather Data	123
4.6.3	JView Based Visualizations for 4D Weather Data	123
4.6.4	User Interface.....	127
4.6.5	Collaboration.....	128
4.7	Wacom Notepad.....	128
4.7.1	Drawing Curved Lines	129
4.7.2	Anti Aliasing	130
4.7.3	Performance	131

4.7.4	Multiple Pages	132
4.7.5	World “Drawer”	133
4.8	3D Model Tools	135
4.8.1	Model Preview Application	135
4.8.2	FLT Model Adjustment Tool.....	136
4.9	JView Demo Browser	138
4.10	3D Models.....	139
5.0	CONCLUSIONS.....	141
REFERENCES		142
ACRONYMS		143

LIST OF FIGURES

Figure 1: A GeodesicBoundingBox with Source Projection	5
Figure 2: Two images sent to the Philips WOWvx display, left: color image, right: depth image	7
Figure 3: Enlarged view of Figure 2, showing header encoding (blue pixels in the top row)	7
Figure 4: Picking Visualization Utility and Bounding Volume Visualization Utility	13
Figure 5: Possible Interpolation Selections.....	17
Figure 6: Camera looking toward a WGS84 projected globe	18
Figure 7: A zoomed-out view of the camera frustum	18
Figure 8: The view footprint of the camera	18
Figure 9: Transfer Function Demonstration.....	19
Figure 10: Two Separate Callout Windows in a JView Scene	19
Figure 11: Labels Element.....	20
Figure 12: JView LabelsElement with drop shadows.....	21
Figure 13: Label 1 placed in the first position tested (North position)	23
Figure 14: A second label is added and the bounding rectangle overlaps the area occupied by label 1	24
Figure 15: All 8 positions are tested until a suitable placement is found.....	24
Figure 16: HUD Text Element.....	25
Figure 17: Histogram Grid displaying surface temperatures from a Lambert-projected data source	26
Figure 18: Sphere Segment Demo	27
Figure 19: Pin Renderer drawing pins with end points from each label to their corresponding locations	28
Figure 20: KML Structure	29
Figure 21: KML Tree Network Links.....	30
Figure 22: KML Tree.....	30
Figure 23: KML Callout	31
Figure 24: JView Scene with KML Placemark Point Scene Elements.....	32
Figure 25: Sample Collada model displayed using JView.....	34
Figure 26: Collada Hand Model with Kinematics	35
Figure 27: Articulated Hand Model	36
Figure 28: Phong shading and illumination applied to a Collada model.	37
Figure 29: Lambert diffuse illumination applied to a Collada model.....	37
Figure 30: Project Health Summary in the Hudson Continuous Integration System.....	39
Figure 31: Icons	40
Figure 32: Node Area with Frustum Culling	42
Figure 33: Camera Distance.....	42
Figure 34: Google Level 21 (~10.8 cm) Satellite Imagery	45
Figure 35: Falcon View Orthorectification.....	46
Figure 36: Tree Based Imagery Selector.....	47
Figure 37: TextureSetTree XML Config File Sample	48
Figure 38: Order Layer Manager	49

Figure 39: World with Texture Masking	50
Figure 40: Bounding Boxes for NAD83 Texture Coverage of an Individual Node	51
Figure 41: Texture Count Before/After	52
Figure 42: Graticule Texture Lines.....	54
Figure 43: World Raster Image Element	55
Figure 44: Grounded SceneElements.....	56
Figure 45: WorldInfoElement.....	57
Figure 46: World Profiler.....	58
Figure 47: World Profiler reports panel.....	59
Figure 48: Reports Panel popup cell.....	59
Figure 49: Triangles Per Node Histogram	59
Figure 50: Textures Per Node Histogram	60
Figure 51: Texture Passes Histogram	60
Figure 52: Invalid Sea Level Post.....	62
Figure 53: Diamond Shaped Difference in ATP.....	63
Figure 54: An Early Development Prototype of the Terra Firma Application	65
Figure 55: Terrain Profile Chart.....	66
Figure 56: Terrain Profile	66
Figure 57: Terrain Profile Callout.....	67
Figure 58: Elevation Projected Line	67
Figure 59: Search Area	69
Figure 60: Texture Set Labels.....	70
Figure 61: All World Features	71
Figure 62: US Features	71
Figure 63: Dragged and Dropped GeoTIFF on JView World in Terra Firma	73
Figure 64: Overview	73
Figure 65: Overview Close up	74
Figure 66: Tabular Scene displaying data from the US Map data source.....	79
Figure 67: Candle stick renderer	80
Figure 68: XY Line and Shape graph	80
Figure 69: XY Block Graph.....	80
Figure 70: XY Bar Graph	80
Figure 71: Pie Graph.....	81
Figure 72: The Polygon Renderer with lighting (left) and without lighting (right).....	81
Figure 73: Comparison of old pre-classification renderer (left), and new pre-integrated renderer (right) ..	83
Figure 74: Comparison of iso-surfaces rendered using the old (left) and new (right) volume renderers ...	83
Figure 75: Anisotropic Scaling of Model Geometry in the Model Renderer.....	85
Figure 76: The Line Renderer showing aircraft trajectories colored based on altitude	86
Figure 77: A Visualization of Regions with Severe Weather Denoted by Weather Polygons	89
Figure 78: A visualization with linear data flow.....	91
Figure 79: A visualization with parallel data flows	91
Figure 80: Jython Code Sample	94

Figure 81: Output from Jython Code Altering Line Color	94
Figure 82: The user interface for specifying a Text Expression	95
Figure 83: An early user interface used for manipulating the Grouping Transform.....	96
Figure 84: The user interface used for editing parameters of the Projection Transform	96
Figure 85: Invalid Entry Indicator	97
Figure 86: Current Tasks window.....	98
Figure 87: The Task Status Icon	98
Figure 88: Docking Frames based User Interface in the ACES Viewer.....	99
Figure 89: Time Control with buttons to disable or pause each controlled table.....	100
Figure 90: The 2D Navigation Control	100
Figure 91: The new wizard user interface.....	101
Figure 92: CSV data import wizard	102
Figure 93: Dataset management panel	103
Figure 94: The Dataset Resolver Dialog.....	104
Figure 95: New Property Sheet User Interface	105
Figure 96: The Schema Editor	106
Figure 97: Reordering Columns in the Schema Editor	106
Figure 98: World Data Configuration Wizard	107
Figure 99: Several graph layouts produced by the new Hierarchical layout algorithm	109
Figure 100: Visualization Composition Graph	109
Figure 101: An Early Version of the Renderers Toolbox, Displayed Using JView's ImageList.....	110
Figure 102: JXTaskPanes that replace the previous JCollapsePanel component	111
Figure 103: The new movie capture user interface.....	112
Figure 104: Table Event Viewer User Interface Component.....	113
Figure 105: An Early Version of the CUNE Viewer	115
Figure 106: Antenna Plots (Point, Line, Surface, Pincushion)	116
Figure 107: Cune Viewer.....	117
Figure 108: Settings Window	118
Figure 109: CUNE Configuration File.....	119
Figure 110: Communications Signal Panel.....	120
Figure 111: The Chart Panel in the CUNE Viewer.....	120
Figure 112: CUNE Viewer on Small	121
Figure 113: CUNE Viewer on Large	121
Figure 114: CUNE Viewer	122
Figure 115: Weather Data on WGS84 World	124
Figure 116: Weather Data on Flat World.....	124
Figure 117: JWeather Cloud Visualization.....	125
Figure 118: JWeather Isosurface Renderer.....	125
Figure 119: JView Visualization of DCF Data.....	126
Figure 120: Visualization of Cloud Profile Along a Flight Path	127
Figure 121: Cloud data from AFWA over the satellite imagery for the same forecast time	127
Figure 122: JWeather User Interface	128

Figure 123: Screenshot of the JWeather Version Distributed to Prologic	128
Figure 124: Bezier Control Points	129
Figure 125: Pixel value under circle	131
Figure 126: Tiled NotePad.....	132
Figure 127: World Drawer.....	134
Figure 128: Model Preview Application.....	135
Figure 129: A model loaded in the ElementViewer, launched from the model previewer application	136
Figure 130: DOF fixer	137
Figure 131: DOF Axis of Rotation	138
Figure 132: Images from Several of the Demo Browser Demonstration Applications	139
Figure 133: Several Models Developed by CACI Rendered with JView.....	140

1.0 EXECUTIVE SUMMARY

The work for this project was divided into the following major areas:

Enhancement of the JView Application Programming Interface (API) – JView is a 2D and 3D, runtime configurable, platform independent visualization API. JView is written entirely in Java and the 3D components utilize the OpenGL API to gain hardware graphics acceleration. Under this contract, CACI provided support to JView users resolving their technical issues, assisting new JView users, and provided general enhancements to the JView API when user requirements were identified.

Development of the "World" Object – Outdoor terrain rendering is important for a large class of Geographic Information System (GIS) applications. Interactive visualization of terrain and general complex surfaces is a difficult problem that requires large data sets to be displayed and manipulated at high frame rates. Operations such as rotation and panning must be supported so a user can examine the data in critical area, while maintaining highly accurate images. CACI developed a Continuous Level-Of-Detail (CLOD) algorithm including Frustum Culling for rendering Digital Terrain Elevation Data (DTED). The CLOD algorithm is based on the observation that 3D objects located far off in the distance may be approximated by simpler versions without loss of visual quality, thus increasing the rendering performance. The "continuous" refers to having the algorithm constantly re-compute the detail level of the 3D object depending on the distance to the camera instead of having a pre-computed set of objects to choose from. This algorithm was used to create the JView World object which has become one of JView's most requested features, and resulted in an Air Force patent by Jason Moore and Aaron McVay. Various performance and efficiency improvements were developed during this effort, including work to support new image sources, to take advantage of multiple hardware processing units (CPUs) and to take advantage of new graphics hardware capabilities.

Research, Explore and Develop De-Cluttering Concepts for Visualization of Overlapping Information – Another area of focused research on this contract was de-cluttering concepts. When there is overlapping information and that information will be visually fused then it must be de-cluttered so that it is intelligible to the intended audience. Presentation of that information is critical to the user's ability to logically correlate it and to come to logical conclusions. An example of this might be an air traffic controller who needs to see a representation of the regions with the greatest density of aircraft. Displaying individual aircraft and their positions would not properly convey the desired information. This project produced several reusable components for JView that reduce information clutter in a visualization.

Research Exploratory Visualization Techniques and Develop an Extensible Application Framework to Support Rapid Integration of New Visualization Capabilities for the Airspace Concepts Evaluation System (ACES) Viewer – CACI continued development of the ACES Viewer application, a tool that provides visualization solutions for displaying and understanding current and future implementations of the National Airspace System (NAS). The ACES Viewer allows researchers to explore a data space by graphically defining a visualization composed of data sources, data processing components, and rendering components. We have extended and enhanced the set of components available to build visualizations while greatly improving the

flexibility of the tool. The ACES Viewer has also been enhanced as a platform for general visualization, allowing developers to extend or replace functionality for a specific domain.

Support of the Audit Trail Viewer (ATV) – The Audit Trail Viewer (ATV) is a replacement for IVIEW 2000 and supports visualization of many different simulation data file formats. The original ATV was developed by the Air Force Research Laboratory's Information Directorate (AFRL/RI) and under this contract, CACI provided maintenance, support, and development of new features for the Audit Trail Viewer.

3D Model Development – CACI developed a library of 3D models in both the Open Flight (.flt) and Wavefront (.obj) model formats for use by JView users. An example model that was developed is the RASCAL 110 aircraft used in the Characterization of the UAV Network Environment (CUNE) project. While there are many models available on the internet for download, most have licensing requirement that make them unusable for JView users.

2.0 OBJECTIVE

The objective of this effort was to develop, implement, and integrate visualization technologies that support NASA and the Federal Aviation Administration (FAA) goals for visualization of the National Airspace System (NAS). CACI collaborated closely with the Air Force Research Laboratory's Decision Support Systems Branch (AFRL/RISB) staff in the development, research, testing, and improvements to the JView API and other visualization tools such as the Airspace Concepts Evaluation System (ACES) Viewer and the Audit Trail Viewer (ATV). Concepts to improve and enhance JView's ability to support NASA's and the FAA's visualization solutions were researched. Non-geographic visualization uses of JView were also analyzed and implemented.

3.0 INTRODUCTION

Under this contract, CACI built on the excellent work done by AFRL/RISB staff and support personnel developing the JView API. JView is a 2D and 3D, runtime configurable, platform independent visualization API. It is written entirely in Java and its 3D components utilize the OpenGL API to gain hardware graphics acceleration.

JView relies on concrete Object Oriented Design (OOD) and programming techniques to provide a robust and venue non-specific visualization environment. There are three types of modules that are created within JView. The first type is venue specific modules called facilitators that address the specific task of placing objects in a scene and manipulating their behavior. The second type are plug-ins that are venue non-specific and add general functionality to the system. The last type consists of data source loaders called oddments. JView is completely data source independent, and is by definition a standard visualization solution since it does not concentrate solely on environments such as space, air, ground, or water. When a new data type becomes available and visualization is necessary, JView allows the programmer to quickly implement the additions. It also allows analysts to operate in an environment in which they are familiar, instead of having to learn a new interface.

In direct support of NASA and Federal Aviation Administration goals for visualization of the National Airspace System, CACI developed a new application using JView technology called the ACES Viewer. The ACES Viewer is an information visualization tool designed to provide visual representations of the output of the Airspace Concept Evaluation System developed at NASA, specifically a simulation of the national air system. The application provides mechanisms to load the various types of data used and output by the ACES simulation, process the data, and display it. The ACES Viewer is not limited to visualization of ACES data; any data (in one of the supported formats) can be loaded and displayed. The application has been enhanced during this effort to serve as a platform for several other domain specific visualization tools.

4.0 TECHNICAL TASKS

4.1 JView

4.1.1 Basic Infrastructure

4.1.1.1 OpenGL Bindings

During the previous JView support contract, CACI migrated JView to the JOGL OpenGL binding library from GL4Java. This library provides the low-level bindings that allow Java applications to access the host platform's graphics drivers (provided as C libraries) and interact with the graphics hardware.

We discovered some problems using JOGL-based applications on 64-bit Windows platforms. One of the native libraries used by JOGL, gluegen-rt.dll, would not load due to an unsatisfied dependency on a Microsoft library that is absent from the default installation of 64-bit Windows. After some experimentation, we found that JOGL could be used without the gluegen-rt.dll on this platform. By modifying the JOGL native library loader utility in JView, and installing the 64 bit Visual C++ redistributable library package from Microsoft (required for JOGL on Win64), we were able to successfully run JView based applications on Windows Server 2003 64-bit, and Windows XP 64-bit. We have not tested JView on 64-bit versions of Microsoft Vista.

We also updated the version of JOGL used by JView to the final 1.1.1 release. The Java OpenGL binding library used by JView to access the host system's graphics drivers has seen significant changes since the initial 1.0 release that was used by JView at the beginning of this effort. After the initial migration to JOGL under the previous contract, we have migrated JView to 1.1.1, and then the 2.0-beta10 version to access new OpenGL capabilities, and improved functionality in several aspects of the JOGL library. We encountered several issues in the newest version of JOGL that we had to work around:

- **Native Library Loading**
JOGL 2 includes some design improvements to the way the platform-specific native libraries are loaded, but the new loading method was not compatible with the way JView previously handled this functionality. As a result, we redesigned JView's dynamic native

library loader to provide more generic capabilities (including support for loading non-JOGL libraries), and integrated this component with the JOGL initialization process.

- **JOGL 2 requires active GLContexts**
Several areas of JOGL 2 now (incorrectly) require a GLContext to be current to obtain information regarding the OpenGL version in use. This includes TextureData (which was specifically designed to support loading of texture data *without* a current GLContext), initialization of GLU functionality (which previously did not require a current context), and static initialization of the GLCanvas class (which imposed a requirement that the JOGL native libraries be loaded before *any* JView drawing primitives could be referenced). We resolved these issues through workarounds in JView where possible, and modifications to the JOGL library where necessary.
- **GLU problems**
Several functions in the GLU library included with JOGL are broken with the 2.0 version. Notable problems include a significant loss of precision in the matrix inversion and multiplication routines used to convert between world and screen coordinates, and enormous memory consumption (4GB when only 10MB should be necessary) in the image scaling routines. We developed replacement functionality for the broken routines in JView.

4.1.1.2 High Precision Locale

Modern graphics processing hardware uses up to 32-bit floating point precision throughout transformation and rasterization processes. Due to the nature of floating point numbers, there is higher precision available for numbers close to zero than for numbers of larger magnitude.

JView has traditionally represented geometry in the scene using a fixed coordinate system. Any location specified as an X, Y, Z triplet uniquely identifies a single location in a fixed coordinate system. In order to view a particular element in JView you move the camera to a location near the element. The problem with this approach is that the camera matrix and the elements position matrix must be multiplied together with only 32 bits of precision to reconstruct the view of the element. Locations specified in a fixed coordinate system will inevitably become large enough (relative to each other) so that the limited precision results in quantization of the locations. This leads to visual artifacts, such as jitter, and distortion of shapes. This has been a significant problem for the World object since its geometry is defined using floating point numbers on the order of 10^6 .

In order to address this problem, we moved part of the vertex transformation process to the CPU, where calculations can be performed with higher precision. This works by defining a dynamically calculated coordinate origin that is a small distance (at the near clipping plane) in front of the camera. All elements in the scene are translated so that their locations are mapped into this translated coordinate system. This translation is performed with double (64 bit) precision. The translated geometry is then sent to the graphics hardware for rendering. Using this

technique, geometry that is near the camera will be rendered with higher precision since it will be translated closer to the origin, with locations near 0,0,0.

We were extremely careful to preserve compatibility with existing JView based applications. For most element types, JView will automatically handle the mapping into the camera-centric coordinate system. A smaller number of elements may need some adjustment to properly take advantage of the additional precision provided by the locale, however they will still work correctly even without modification.

We updated all of the JView rendering components (Scene Elements) that would benefit from the greater precision available with this enhancement, particularly those intended for use with JView World.

4.1.1.3 Geodesic Bounding Box with Source Projection

The Geodesic Bounding Box is a JView component, originally developed for the ACES Viewer, used to represent a geographic region. Previously, this component was only able to represent projection-agnostic regions, in other words a region was specified in terms of latitude, longitude, and altitude, and a corresponding span for each dimension. We have enhanced this component to additionally support projection dependant regions, as shown in Figure 1. A GeodesicBoundingBox defined as a rectangle in a Lambert Conformal projection (left) is curved in a Mercator projection (right).

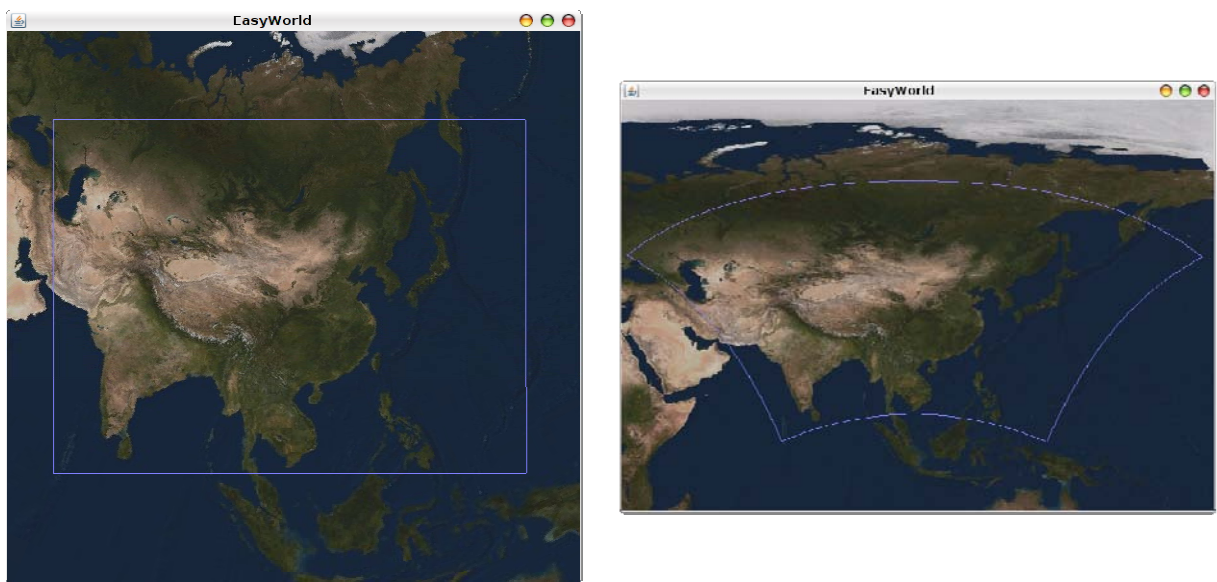


Figure 1: A GeodesicBoundingBox with Source Projection

4.1.1.4 Stereoscopic Rendering

We made several improvements to the stereoscopic rendering support in JView. First, we restored support for stereoscopic visualization using JView in both combined anaglyph and separate stereo views. While the infrastructure for stereoscopy has been present in JView for a long time, this functionality was broken during the switch from GL4Java to JOGL as the underlying OpenGL binding layer.

To improve the efficiency of certain JView scene elements that use the camera frustum for visibility culling and level of detail calculations when viewing the scene in stereo mode, (most notably the World element) we implemented calculation of a combined frustum that encloses the individual frustums for each rendered point of view.

4.1.1.4.1 Philips WOWvx Display

We incorporated support for the Philips WOWvx auto-stereoscopic display into JView. This display device allows 3D visualizations to be rendered in stereo by transmitting different images to each of the viewer's eyes using specialized lenses and filters. In order to do this, the display requires the 3D scene to be transmitted to the device in a special format that specifies the color and depth of each image pixel, along with some parameters that control display characteristics of the device.

Pre-existing support for stereoscopic display hardware in JView worked by sending two separate color-only images of the scene (one for each eye) to the device. For the Philips device, we also send two images, one being a color image of the scene. The second image is a grayscale disparity map that essentially describes the distance to each object that is visible in the scene. The disparity map can be calculated from the values in OpenGL's depth buffer after rendering the color image. Using a combination of off-screen rendering to frame-buffer objects, and GLSL shader programs we were able to render the two images side by side (Figure 2) with very little performance impact vs. just rendering a single color image of the scene.

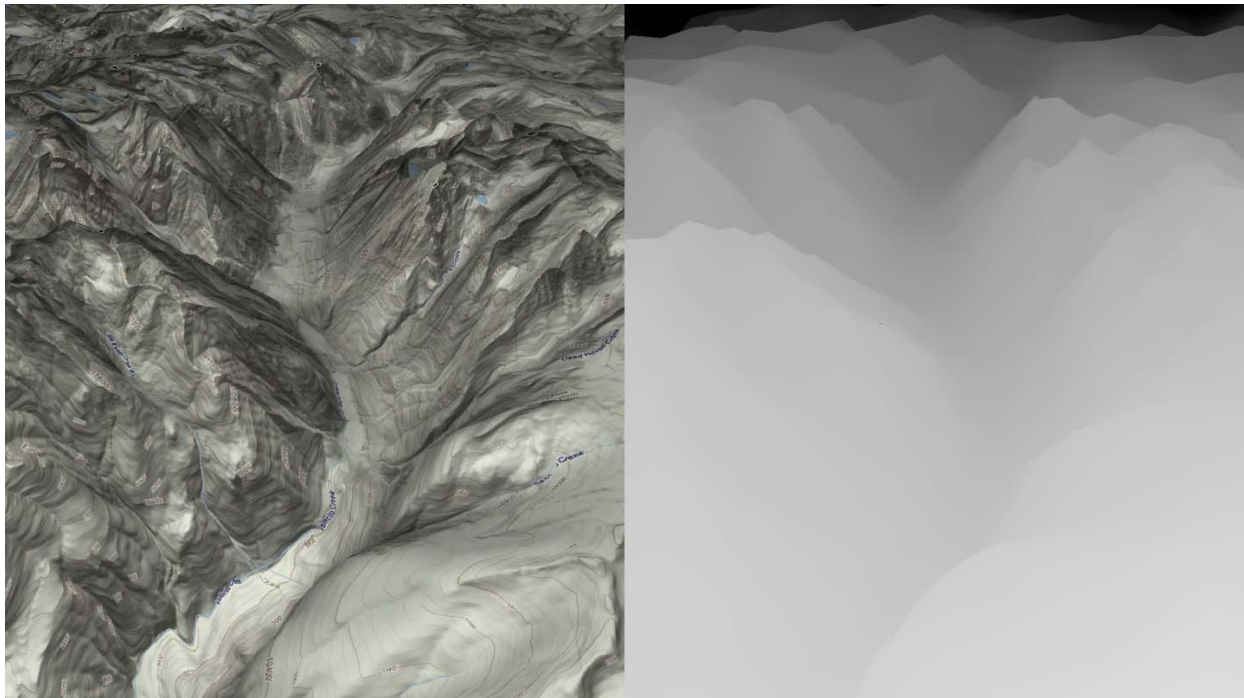


Figure 2: Two images sent to the Philips WOWvx display, left: color image, right: depth image

In addition to the two images, the device also requires a specialized header to activate the stereoscopic display mode and configure the display properties. The header data is encoded in the image, in the blue channel of the first row of pixels. We incorporated this information using another GLSL shader, as shown in Figure 3.



Figure 3: Enlarged view of Figure 2, showing header encoding (blue pixels in the top row)

4.1.1.4.2 Quad Buffered Stereo

CACI developed support for quad-buffered stereo drawing surfaces into JView. Quad Buffered stereo renders a 3D scene into two separate image buffers (compared to the typical single buffer for normal rendering), one containing an image for a left eye, and one for a right eye. Compatible display devices (such as a fast refresh monitor with shutter glasses) allow the left and right eye images to be presented to the corresponding eyes of the viewer. The components that were developed allow virtually any JView based application to render arbitrary scenes in this mode. Quad Buffered Stereo support has not been a high priority in JView, and while it is currently capable of properly rendering images for both camera viewpoints, there appears to be a problem activating the stereo viewing mode on the NVidia NVision display that we were testing this functionality. This is an area that will be addressed.

4.1.1.5 GLSL Shader Program Abstractions

In order to support the use of OpenGL Shading Language (GLSL) programs in JView for the Philips WoWvx display and other components, we developed several utilities to manage, load, and activate shader programs. These utilities provide an extremely simple API for working with shader programs, and can be used with any GLSL programs. They provide functionality that includes:

- Loading programs from disk
- Compilation from GLSL source code
- Activating and de-activating programs
- Resource management and integration with Java's garbage collection system
- Program validation

4.1.1.6 OpenGL Lighting

JView has always included support for a single directional light source that is on by default. While some properties of the light source can be controlled via methods on Graph3D, this required some knowledge of low-level OpenGL lighting functionality and required developers of JView-based applications to explicitly set the parameters. We have added a new capability to Graph3D to control some of the lighting parameters automatically, based on the camera position. With one or two simple method calls on Graph3D, users can cause the light to track the camera position and make it appear that the light is originating from behind the user. Users may also select directional or positional lighting for different effects.

4.1.1.7 Geographic Coordinate Systems

JView's Coordinate API was originally designed with two coordinate representations in mind: Cartesian (X,Y,Z) coordinates and flat-earth geodesic (Lat, Lon, Alt) coordinates. Development of the World object resulted in development of the Projection API and several new projections (Lambert Conformal, WGS84) that were being commonly used with JView. The existing flat-earth geodesic coordinate classes were not suitable for non-flat-earth projections, which in turn led to development of a new geodesic coordinate implementation for the WGS84 mapping. This led to various difficulties faced by application developers, who had to decide which coordinate class to use based on the projection being used for the scene. To simplify use of the Coordinate API, we implemented a Projected Coordinate class that can work for any geodesic coordinate system with an associated Projection implementation. A further advantage of the new Coordinate is that the Projection can be changed on the fly, without creating a new Coordinate.

4.1.1.8 Rendering Appearance Settings

JView uses a component called the Appearance Processor, and an associated AppearanceSettings value class to track and change various OpenGL rendering settings. This component is typically used in the geometry tree part of the JView scene-graph to manage per-element and per-geometry node settings. We have improved this component by introducing a new method to change the OpenGL state based on the AppearanceSettings instance that represents the current settings, and another AppearanceSettings instance that represents the desired state. This provides a performance improvement in some cases when settings need to be changed multiple times when rendering a single element. The previous behavior was to apply new settings in a similar manner. However, the original settings were always restored before changing them again even if the original settings were not needed any more.

We added the ability to specify a modified depth range used when rasterizing geometric primitives. The depth range specifies the range of floating point numbers that will be used to represent the normalized depth of each element in the scene. Due to the way that floating point numbers are encoded, there is greater precision available for values near 0, which is traditionally mapped to the depth of the near clipping plane, giving higher depth precision closer to the viewer. In some cases, it is desirable to invert the depth range, mapping the far clipping plane to 0, giving higher precision in that area. With the changes, AppearanceSettings can now solely use JView API calls (as opposed to OpenGL calls). This setting also provides a means of manipulating values that are placed in the depth buffer (used to determine which objects are closer to the image plane). Compressing the depth range (particularly by lowering the far distance) allows elements to be ‘nudged’ slightly closer to the camera location, ensuring they appear in front of other objects in the scene.

4.1.1.9 Animation Framework

We incorporated a new animation framework that was initially developed for the ACES Viewer into JView. The new framework serves as a replacement for the AnimationManager and the problematic TCManager systems in JView, in addition to providing new capabilities and improvements. One feature of note in the new system is the modular timing system. Animation timing can now be easily linked to the rendering frame rate, or a non-real time clock (for movie capturing), in addition to using the system clock.

We also re-factored the animation framework after moving it into JView, to provide a more complete and easier to use replacement for the original animation and time constrained simulation playback subsystems. This work included:

- Implementation of a simulation playback animation that operates on fixed precision timing values.
- Implementation of a simulation playback animation that operates on floating point precision timing values for better performance where fixed precision is not necessary.
- Refactoring of the Animation and TimingSource APIs to reduce complexity. This allows developers to implement animated behaviors without having to understand the underlying dynamics of the animation system.

- Ensuring that correct timing information is provided to animations. For example, we ensure that time is not lost due to floating point roundoff.
- Added support for cumulative timing (the amount of time elapsed since the animation was started) in addition to the existing incremental timing (time since the previous animation frame).
- Re-implemented the existing TCManager subsystem (for simulation playback) on top of the new animation framework while preserving compatibility with applications that used it. The TCManager system was moved out of JView, but is still used by several applications that cannot migrate to the new API for various non-technical reasons.
- Migrated all existing JView components to the new API.

4.1.1.10 Vertex Buffer Objects

Vertex Buffer Objects (VBOs) are a relatively new OpenGL data structure used to represent geometric data for rendering. We developed a VBO abstraction for JView that simplifies their usage.

VBOs were designed as a replacement for older methods of sending this data to the graphics hardware, including immediate mode, display lists, and vertex arrays. From a low level perspective, VBOs are simply regions of memory allocated by OpenGL that are typically directly accessible by the graphics hardware (e.g. video memory or AGP memory). An application will fill one of these regions with data, and subsequently bind it for rendering.

Minimally, to render an image, a VBO must contain vertex data. For each vertex in a buffer, additional information can be provided, including colors, normals, texture coordinates, etc. The binding process involves describing what regions of the buffers contain each type of data. These regions can be stored in a single VBO, or can be split among several buffers. This leads to the three storage models supported by OpenGL and our abstraction:

- Data stored sequentially in a single buffer. For a buffer containing vertex (V) and color (C) data, this might be stored in a single buffer with all similarly typed data grouped together: [VVVCCC]. This layout is probably the easiest to manage.
- Data stored interleaved in a single buffer. By alternating the vertex data with any attribute data associated with each vertex, data that will likely be used together will be stored together: [VCVCVC]. This can lead to significant performance improvements, particularly for large buffers.
- Data stored in multiple buffers. In some circumstances, you may want to separate the vertex data and some or all of the vertex attribute data. For example, you might frequently update vertex colors, but never update their locations. In this case, you can allocate two separate VBOs, and via hints to OpenGL, you can request that one buffer be used for static content, and one used for dynamic or streaming content. The graphics driver will decide where to allocate each buffer (VRAM, AGP, or DRAM) for the most efficient access. With this model, you would have [VVV] [CCC].

To simplify usage, and reduce the potential for programming errors, we track which buffer is active on a thread at any given time, and which buffers are bound for rendering. Using this

functionality, we are able to automatically enable or disable, and bind or unbind a VBO with a single method call from the user.

Our abstraction contains a buffer layout specification that allows users to specify what type of data should be stored in each buffer, and how it should be organized. The buffer layout contains or computes all of the necessary information to initialize a buffer object, bind regions for drawing, or to provide pointer offset information for direct user manipulation of the buffer content. The layout abstraction is extensible, so that if new data types are added to the OpenGL VBO specification, support can be easily added to our abstractions.

Similar to how texture data is managed in OpenGL, each VBO is assigned a unique numeric identifier when it is created. Since the memory used by the VBO is outside of the Java heap space, it is not directly managed by Java's garbage collection system. This can lead to memory leaks if a VBO is not explicitly deleted when it is no longer in use. Our VBO abstraction ensures that the VBO it represents is deleted when the Java memory associated with the abstraction classes is garbage collected. We also provide a means to explicitly delete the VBO for more user control over the timing of the operation.

4.1.1.11 Texture API

We developed a new Texture API that is both more efficient and more flexible than the previous systems available in JView. The new Texture API has a number of advantages over the earlier TextureContainer system:

- Improved configurability of texture environment settings and texture parameters without requiring users to make explicit OpenGL calls.
- Support for multiple texture formats (1D, 2D, 2D non-power-of-two, 3D, etc.)
- Support for the full range of pixel formats supported by OpenGL.
- Simplified usage. After initial configuration of the texture, users only need to call enable or disable on the appropriate texture, or use JView's AppearanceSettings system to automatically enable or disable textures.
- Thread safe updates. The previous texture system required changes to the texture to be performed in the OpenGL rendering thread, while the new system allows changes (such as image updates) to be initiated from any thread.

The new texture system also enabled development of support for 'live textures', textures that represent a frequently changing image source such as a Java Swing UI component, or a video under several other contracts.

4.1.1.12 Picking/Selection

We enhanced the CPU based picking subsystem that was developed under the User Defined Operational Picture effort and incorporated it into JView. Previously, JView relied on the system

graphics drivers (via the OpenGL API) to handle picking in a 3D scene. This approach had several drawbacks:

- OpenGL based picking has been removed from newer versions of the OpenGL API. We have also noticed that picking has become slower in each new release of the graphics drivers from major vendors, as it becomes more difficult to implement with the increasing programmability of the graphics pipeline.
- OpenGL based picking operates on geometric primitives, and can only provide information on which primitive was picked. This can cause difficulty in certain situations where a single primitive represents multiple pieces of data in a model. For example, a complex user interface component can be texture mapped onto a single OpenGL quad. Additional application specific code is required to translate the OpenGL selection information to determine which control might have been selected.
- The result of a picking operation in OpenGL consists of a set of integral identifiers and depth information. This information needs to be programmatically mapped back to the model object represented by each identifier.

The new system that we developed abandons OpenGL picking APIs in favor of a custom Java API that has been incorporated into the core rendering subsystem in JView. We provide default implementations of this API for all renderable components in JView, but allow developers to override or extend the default behavior to introduce domain specific data objects to be picked in addition to the renderable components.

We also provide a concise picking utility API to assist developers that wish to implement custom picking for their renderable components, and to interpret the data provided in response to a picking action. These utilities include a simple to use picking visualization component that can be added to a 3D scene with a single line of code to provide visual feedback indicating where picking ray intersections occur. An example of this utility is shown in Figure 4. The Picking Visualization Utility is the pink line with yellow dots indicating intersections, and the Bounding Volume visualization utility is represented by the purple boxes.

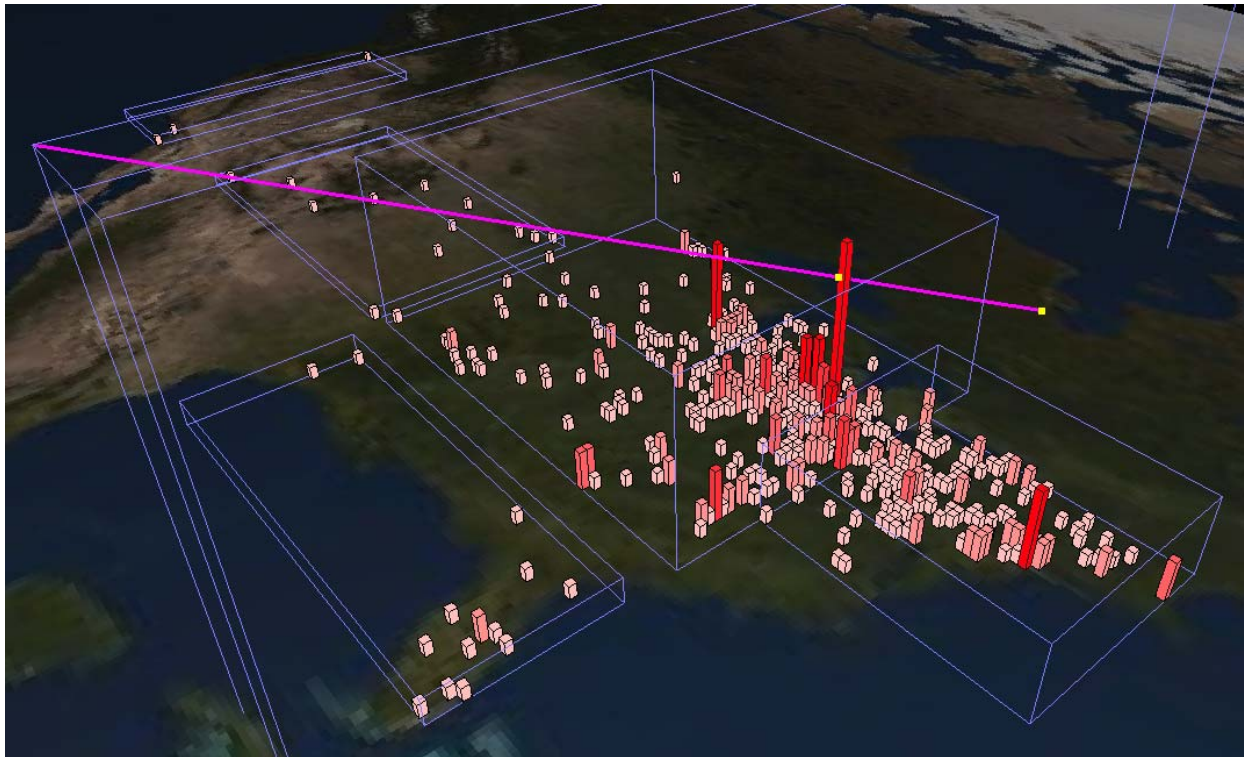


Figure 4: Picking Visualization Utility and Bounding Volume Visualization Utility

4.1.1.13 Bounding Volumes

Many scene graph libraries associate a bounding volume with each renderable object to provide a consistent and simple mechanism to determine an approximate area where the object will be drawn. This information can be used for automatic frustum culling, clipping plane management, navigation (determining where the camera should be positioned to view certain elements of the scene), and other common functions. Earlier versions of JView provided an optional API for scene elements to provide this information, however since the engine could not depend on bounding volume information being present we could not take full advantage of the benefits it could provide. Additionally, since it was optional and not an integral part of the core JView API, it was difficult to use, and many developers were not aware it was available.

Each scene element in JView now has an associated bounding volume. For scene elements that are constructed completely from the geometric primitives included with JView, the bounding volume is computed automatically, and requires no additional work on the part of the developer. For custom scene element implementations, we provide a simple API by which the developer can specify a bounding volume in the object space of the element it describes. We provide three bounding volume implementations for this purpose: axis aligned bounding box, oriented bounding box, and bounding sphere. Developers are free to provide their own implementations as well.

JView's scene graph system automatically converts the volumes to world coordinates (which automatically handles scene element coordinate changes), and composites them with volumes from other scene elements to produce a bounding volume for the entire scene. This is done in a manner that preserves the original volumes, rather than computing a single volume that encompasses all sub-volumes, which is important for certain functionality that relies on tight bounding volumes for optimal behavior (such as clipping plane management).

With bounding volume information integrated into all JView based scenes, we were able to improve and simplify several other subsystems, including clip plane management, navigation, and frustum culling. We also provide a utility that can display the bounding volumes in-scene with a single method call, primarily useful for debugging. An example of this visualization is shown in Figure 4.

4.1.1.14 Camera System

JView's Camera abstraction is designed to allow developers to manipulate viewing parameters such as position and orientation in a more intuitive manner than directly manipulating OpenGL's transformation matrices. We made several improvements to the Camera that improves usability, performance, and accuracy.

4.1.1.14.1 Removed Dependency on GL Utilities Library

After migrating JView to version 2.0 of the Java OpenGL (JOGL) binding library, we discovered some significant changes that negatively impacted the mathematical precision used for viewing transformation calculations. We addressed these problems by implementing our own versions of the relevant functionality. These changes provided other benefits to JView in that more information is now available to apply viewing transformations to geometry with a relatively simple API. This API is used by a variety of renderers to improve performance (e.g. frustum culling), functionality (e.g. picking), and to simplify implementation.

4.1.1.14.2 Clipping Plane Management

Clipping planes are used in rasterization based 3D graphics systems to limit the renderable depth range to maximize depth buffer precision. Originally, JView required developers to manually specify the depth range for a given camera position and scene content. Each time the camera was moved, or the scene content changed, the clipping plane distances had to be re-specified. We previously automated this process using an ancillary API that was built in to one of the camera navigation systems in JView. During this period, we incorporated automated clipping plane management into the core render loop in JView.

In the new API, clipping plane distances are calculated from the bounding volumes of the scene elements by default, at the beginning of each render pass. This means that unlike the previous implementation, developers using JView do not have to do anything explicit to integrate

their scene elements with the clipping plane management system. The default behavior can easily be overridden to customize the clipping distance calculations for each element. The new system supports a combination of hard and soft distance constraints, which allows JView to determine the optimal clip distances for a scene, balancing depth buffer precision against the desire to render all visible elements for a given camera position.

4.1.2 Utilities

4.1.2.1 Native Library Loader

During the previous JView support contract, we developed utilities to simplify the process of using JView and JView based applications by packaging and dynamically loading the platform specific C libraries required to access OpenGL functionality. During this effort, we improved this functionality by adding support for multiple copies of the C libraries to coexist within a single system (different JView applications may use different library versions). Furthermore, we made the dynamic extraction and loading utility more generic so that it can be used for other third-party C libraries (such as those used by Java Advanced Imaging and JInput).

After an update the version of JOGL used by JView, we discovered a problem using JView based applications on systems that had run applications with an earlier version of JOGL. The problem was that an older version of JOGL was still present on these machines in the shared jogl-natives temporary folder. This caused the application to crash when attempting to load the incorrect version of the native library. To address this problem, we incorporated a workaround for a Windows specific issue into the Native Library Loader. Due to file locking semantics on Windows based systems, the native libraries are now deleted by a separate process that is launched when the original application exits.

4.1.2.2 Camera Navigation

We enhanced the camera navigation framework that we had previously developed for JView by adding an interface for components that determine the coordinate of the intersection of mouse events with the elements in the scene. We have developed three implementations of this interface, to detect intersections with SceneElements (with selection enabled), the map projection surface, and the JView World element respectively. This functionality is used when performing pan operations with the mouse to keep the hit object directly under the mouse cursor for the duration of the operation, and to re-target the camera in response to a double-click.

We also improved the usability of the navigator by allowing other motions to be active during a transition. For example, the user can now modify the elevation or azimuth of the camera relative to the moving anchor during a transition. Previously, transitions were exclusive operations.

We further simplified the navigation system by modeling the camera's position and orientation with vectors and quaternions. More specifically, we use a vector and quaternion to represent the location and coordinate-system orientation of the look-at point, and another

vector/quaternion to represent the camera's relationship to the look-at point. This is similar to earlier representations, but implemented more efficiently.

We improved and simplified the navigation system's API and internal functionality by using quaternions to represent orientation. This reduces the number of 'special cases' (of orientation or position), reduces the number of calculations necessary to perform common operations, and increases the precision of most rotation calculations over other representations (matrices or Euler angles).

4.1.2.2.1 Bounding Box Position Constraint

We implemented a bounding box based camera position constraint for use with JView's camera navigation system. This constraint allows developers to limit the look-at point for the navigator to a region defined by a geodesic or axis-aligned bounding box. This constraint is particularly useful when viewing the world with a flat projection, allowing developers to prevent the camera from moving off the edge of the map, but can also be used to constrain the viewing position to a region of interest.

4.1.2.2.2 World Orbit Navigator

We added functionality to the World Orbit Navigator (a JView World specific extension of the camera navigation system) to take terrain elevation into account when orbiting around the navigator's anchor location. Previously, the camera typically orbited around a (moveable) point on the map projection manifold, defined by the anchor location. In areas where there were large variations in terrain elevation, this behavior produced unexpected results, because the anchor location was typically located beneath the terrain. The improvements compute the intersection of the camera's viewing ray (through the center of the screen) with the terrain each time the anchor point is moved, and automatically adjust the anchor to the location of the intersection. This is done without moving the camera, so from the user's perspective, the change in position of the anchor is seamless, however when the user initiates an orbit motion, they now orbit around the surface of the terrain, rather than the surface of the map projection manifold.

4.1.2.3 DTED Interpolation

A new class called *Utilities* for the mil.afrl.rrs.ifs.b.dted package was developed that includes a `getElevation` method that returns an interpolated elevation. Any given longitude/latitude point will lie inside a quad described with a southwest corner (LOWER_LEFT), northwest corner (UPPER_LEFT), northeast corner (UPPER_RIGHT), and southeast corner (LOWER_RIGHT). Figure 5 shows the possible interpolation selections, and if no data is available from the DTED server, this method returns DTED_NULL

Interpolation Method	Returned Value
LOWER_LEFT_POST	southwest DTED post will be returned
UPPER_LEFT_POST	northwest DTED post will be returned
UPPER_RIGHT_POST	northeast DTED post will be returned
LOWER_RIGHT_POST	southeast DTED post will be returned
NEAREST_POST	geometrically nearest post (the exact center will return the southwest post)
NEAREST_LL_UR_POST	imagine a quad split into a lower left triangle and upper right triangle, will return geometrically closest corner of the appropriate triangle
NEAREST_UL_LR_POST	imagine a quad split into an upper left triangle and lower right triangle, will return geometrically closest corner of the appropriate triangle
BILINEAR	weighted average using all four corners
TRIANGULAR_LL_UR	weighted average using the appropriate three corners of the enclosing triangle
TRIANGULAR_UL_LR	weighted average using the appropriate three corners of the enclosing triangle

Figure 5: Possible Interpolation Selections

4.1.2.4 View Footprint Calculator

We developed an algorithm to calculate the *view footprint* of a camera in terms of the boundary between the visible and non-visible latitudes and longitudes. The footprint is calculated by sampling the intersections of viewing rays from the camera location through the edges of the viewport at regular intervals. For each sample, the ray might intersect with the surface defined by a map projection or not. Where there is such an intersection, the latitude and longitude of the intersection is incorporated into the view footprint outline. When there is no intersection, we calculate the latitude and longitude of the horizon in the direction of the viewing ray.

The view footprint algorithm is used by the ACES Viewer for its 2D camera position controller and by Terra Firma for the overview component. A sample scene is shown in Figure 6 with the camera looking toward a WGS84 projected globe. Figure 7 shows a zoomed-out view of the camera frustum for the viewpoint in Figure 6. Figure 8 shows the view footprint of the camera in Figure 6. The upper-most section of the green footprint outline corresponds to the horizon, while the side and bottom edges result from the intersection of the frustum with the surface.



Figure 6: Camera looking toward a WGS84 projected globe

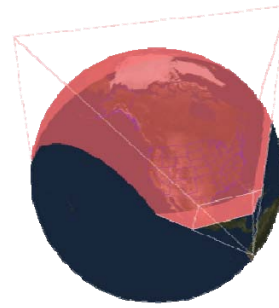


Figure 7: A zoomed-out view of the camera frustum



Figure 8: The view footprint of the camera

4.1.2.5 Movie Capture

We transitioned the Movie Capture utilities developed for the ACES Viewer into JView. These utilities provide a higher level API to work with JView's existing movie capture subsystem that will simplify the process for most users. Major functionality includes encapsulation of movie parameters (encoding, frame rate, watermark, output file, etc.), integration with JView's animation framework, and management of the capture process. Users typically only need to implement a simple callback interface to update the view between captured movie frames, everything else is handled by the new utilities.

4.1.2.6 Transfer Function Editor

JView provides several implementations of a Transfer Function, a function that maps numeric values to visual properties (typically color and opacity). These functions can be used by any visualization component that requires such a mapping. We recently re-factored the TransferFunction API to improve its usability and to simplify implementation of new Transfer

Functions. We also developed a user interface component to display and manipulate the properties of a Transfer Function.

The Transfer Function Editor provides a visual representation of the colors and opacities produced by the transfer function over its input value range. Users can interact with the display to manipulate the mapping, and thus affect the visualization using the Transfer Function. An example of the user interface is shown on the left side of Figure 9, and the resulting Transfer Function applied to an array of spheres (right side).

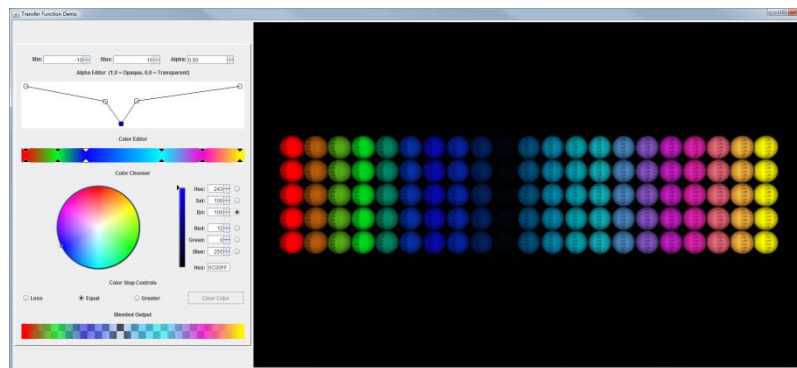


Figure 9: Transfer Function Demonstration

4.1.3 Rendering Elements

4.1.3.1 Callouts

We developed a new Callout implementation that creates a 2D SceneElement window (optionally with window decorations) that presents a Java2D graphics API for easy rendering to the window. An optional connector can be drawn from the window to a location in 3D space. A WorldCallout is also available that's World aware that can draw the connector to a lat/lon on the World and disable the WorldOrbitNavigator for window drags (Figure 10).

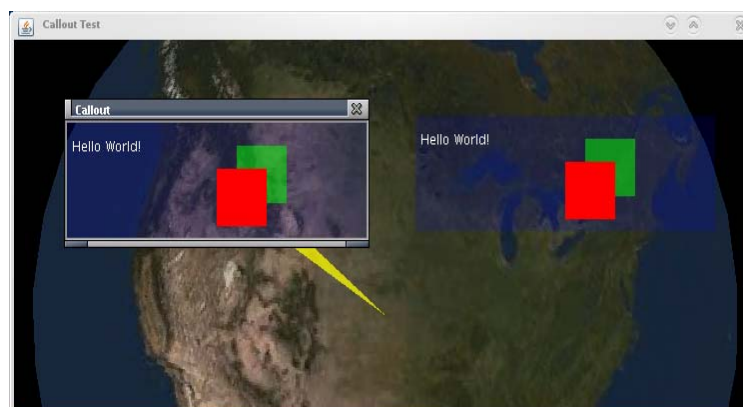


Figure 10: Two Separate Callout Windows in a JView Scene

The callout system was used as the basis for an improved version that was developed under a different contract that was only possible after we implemented support for rendering Java Swing components in a JView/OpenGL scene.

4.1.3.2 Labels Element

We incorporated a variety of improvements to the Labels Element (Figure 11) that was originally developed for the ACES Viewer application during the previous JView support contract, and migrated to JView during this effort.

Of note is the addition of per-label visibility priorities. This provides control to a developer over the de-cluttering process. Each label in a scene has an associated numerical priority. Elements with a high priority will never be considered occluded by a label with a lower priority. This gives developers the ability to ensure that certain labels are visible even if they are occluded by other 'less important' labels. Labels with identical priority values obey the normal depth-based de-cluttering process.

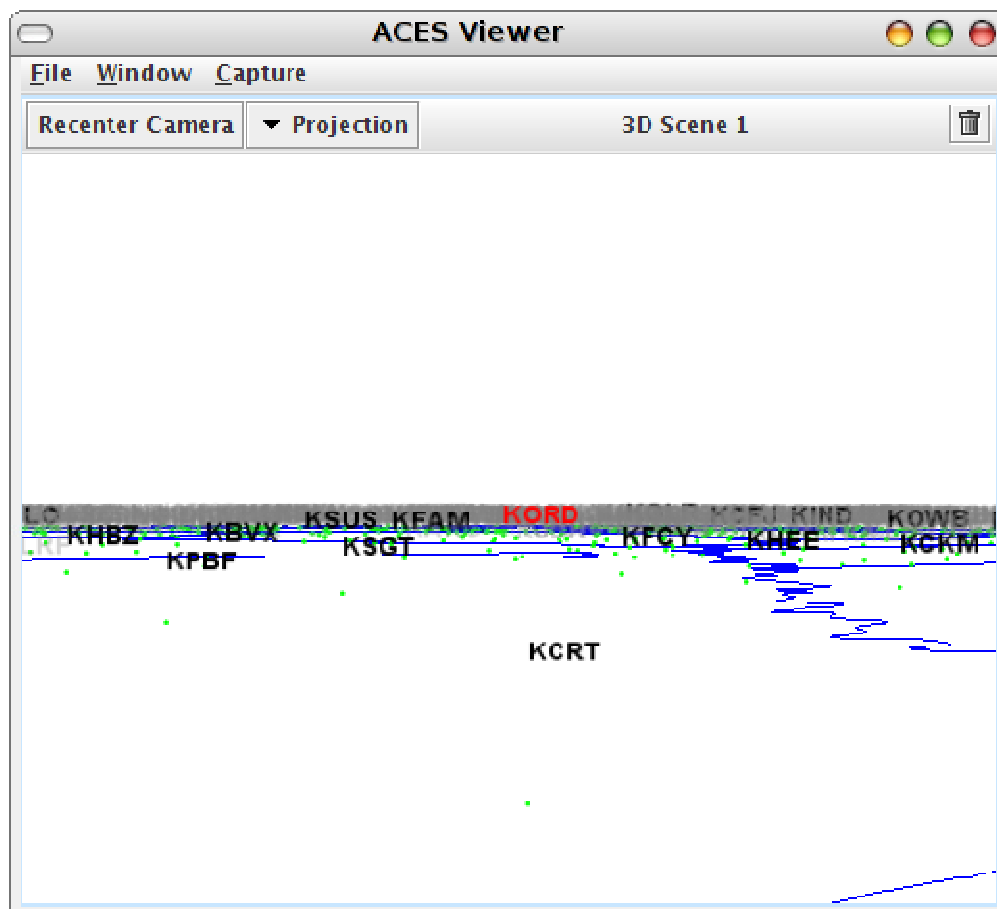


Figure 11: Labels Element

We have also added the ability to set the text color on a per-label basis, line justification (left, centered or right aligned), improved alignment configuration settings, and support for external occluders (multiple LabelsElements instances occluding each other). Users can also enable drawing of drop-shadow style outlines for the label text to improve legibility in cluttered scenes (Figure 12).

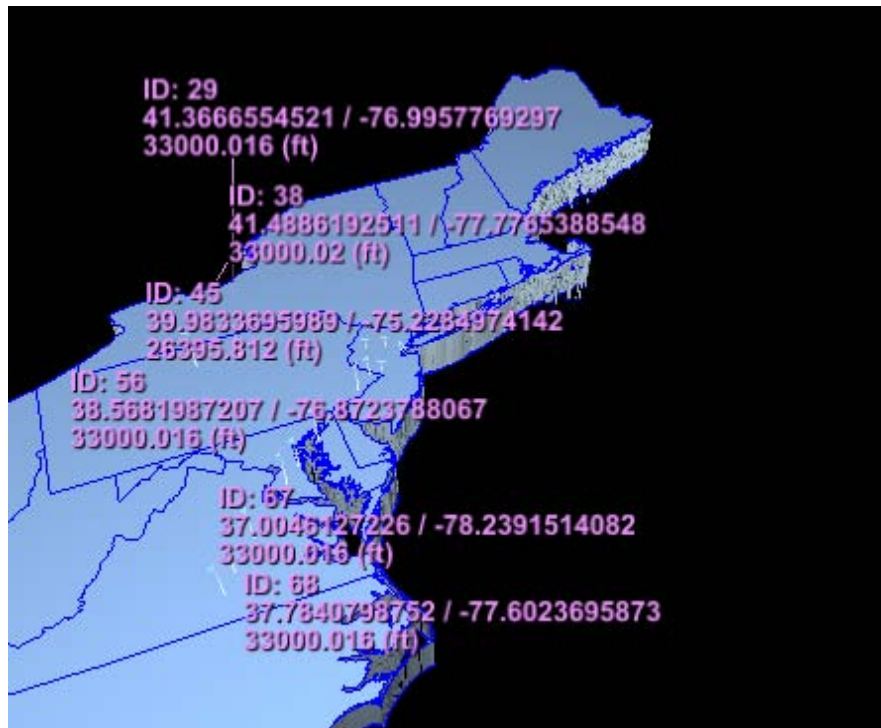


Figure 12: JView LabelsElement with drop shadows

4.1.3.2.1 Automatic Label Decluttering with Dynamic Placement

We have improved the de-cluttering algorithm used by the textual label renderer in JView to include automatic placement of the label in a user defined area around the object being labeled.

The original de-cluttering algorithm determined whether a label was occluded by another label closer to the camera viewpoint using a K-D Tree based occlusion query data structure. If a label was determined to be occluded, its color was changed so the occluded label was less visible, and did not interfere with the legibility of the occluding label.

Based on research presented at the 2008 InfoVis conference, improvements were made to the algorithm to increase the number of labels in a non-occluded state, by allowing the occluded labels to be re-positioned in a small area around the object the label is associated with. Our implementation of this algorithm (with some enhancements) works as follows:

1. Calculate the screen area of all labels (represented by a 2D rectangle). Cull any labels whose associated object (the item being labeled) does not fall within the viewing area.

2. Sort the remaining labels. Labels that are positioned earlier during placement and occlusion testing will be less likely to be occluded by another label. We use a combination of priority based and depth based sorting to achieve a suitable ordering.
3. Calculate a screen location for each label (that has not been placed in a non-occluded position) using user-defined minimum and maximum distance constraints, and placement algorithm.
4. For each label that was positioned in step 3, query a global occlusion structure to determine if the label overlaps another object in the occlusion structure. If there is no overlap, commit the label's position, and mark the corresponding region in the occlusion structure.
5. Repeat steps 3-4 for all labels that have not been successfully placed, using alternate placement algorithms until all labels have been placed, or all placement algorithms have been used.
6. Finally, any labels that have not been successfully placed are marked occluded, and their color is adjusted in the same manner as the original de-cluttering algorithm.

Our improvements to the original placement technique include animated transitions between placement locations and occluded state (color). We also improved the coherency of the label location relative to the labeled object while the object is moving on-screen (either by motion of the object, or changes in the viewing direction/position). This was accomplished by detecting changes in the labeled object's screen location and only re-positioning the label if the location has not changed for a specified amount of time. Occlusion detection is still performed even if the labeled object's position has changed, so while labels associated with a moving object will not be repositioned, they will still fade in and out, as other labels occlude them.

This algorithm depends on two other components in JView, the occlusion query data structure, and the label placement algorithm(s).

4.1.3.2.2 Occlusion Query Data Structure

The Occlusion Query data structure used by the original de-cluttering algorithm was too slow to handle the significantly larger number of queries issued by the new de-cluttering algorithm. Previously, only one query was issued per label to determine whether it was occluded. The new algorithm requires a query for each label, at each position tested, which can easily result in 100 times as many queries in a scene with a large number of labels. To improve performance, we implemented a new data structure with $O(1)$ running time for queries.

The OcclusionStructure represents the viewport area as a 2D grid of bits. Each bit represents a small area on the screen, which could be as small as a single pixel, but could be larger for more coarse-grained occlusion testing with better performance. Bits set to 1 indicate a region that is occupied, and bits set to 0 represent un-occupied space. The data structure provides methods to test areas of the grid to determine whether any bits are set in a specified rectangle, and to set bits in a specified rectangle. This representation requires somewhat more memory than the previous K-D Tree based data structure when there are a small number of occluders, but scales more efficiently to handle a larger number of occluders.

In addition to the performance improvements, we associate a single instance of the Occlusion Query data structure with each scene. A scene's Occlusion Query structure is automatically cleared at the beginning of each render pass, and can be shared by all elements participating in the rendering of the scene (i.e. other non-label objects can mark regions as occupied). This permits detection of other occluders, in addition to just label-label occlusions.

4.1.3.2.3 Label Placement Algorithms

Label placement is determined by a sequence of user-specified placement algorithms. Each algorithm is applied to all unplaced labels in sequence until all labels have been placed, or the placement algorithm sequence has been exhausted. We have implemented three algorithms that are used together as the default placement algorithm sequence.

- Eight-Position placement. Labels are placed in one of eight positions in a circle around the labeled object corresponding to the compass directions N, NE, E, SE, S, SW, W, and NW. The first position in this sequence corresponding to a non-occluded position for the label (if any) is used. Diagrams of this placement algorithm working are shown in Figure 13, Figure 14, and Figure 15.
- Spiral placement. A sequence of label positions following a spiral pattern around the labeled object, with configurable distance range, spiral rotations, and number of locations to test. The default label placement sequence uses two instances of this algorithm with different parameters.
- Default Placement. This places the label directly above the labeled object, regardless of whether it is occluded at that position. This is used to place all labels that were not successfully placed by another algorithm. This can also be used to emulate the original de-cluttering algorithm by placing all labels at a fixed location relative to the labeled objects.

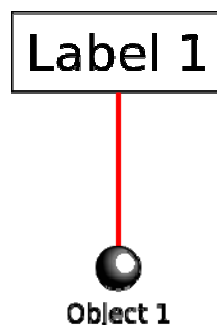


Figure 13: Label 1 placed in the first position tested (North position)

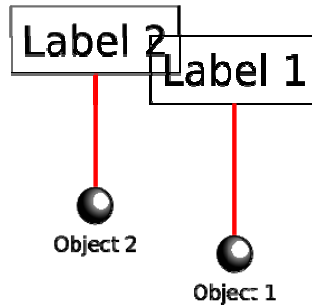


Figure 14: A second label is added and the bounding rectangle overlaps the area occupied by label 1

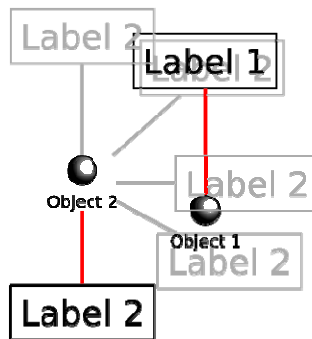


Figure 15: All 8 positions are tested until a suitable placement is found

Users can also implement and use their own placement algorithms if necessary to achieve the desired placement behavior.

4.1.3.3 Instance Rendering

Initially developed to support the need of the ACES Viewer application to render thousands of aircraft models at different locations, we developed Instancing support in JView. Instance rendering refers to the ability to prepare the graphics hardware for rendering an element once, while actually drawing it in multiple locations. This amortizes the overhead of sending geometry and appearance information to the graphics hardware over multiple instances of the geometry.

Initially, we implemented this using OpenGL display lists that allowed the geometric data to be transmitted to the graphics hardware once, and then be rendered at multiple locations. Tests with several types of JView-provided geometry show significant (2x-10x) performance increases when rendering many instances of the geometry using this approach.

More recently, as the necessary OpenGL functionality became more widely available on computer systems, we were able to further improve performance using hardware instancing techniques. This approach was also developed for the ACES Viewer and migrated to JView after some stabilization. Hardware instancing takes advantage of the programmable graphics pipeline, and newer OpenGL data representation and transfer abstractions (buffers) to render geometry at many locations with very few OpenGL calls. The reduction in communication overhead between the CPU and GPU (by making fewer calls) achieves another 10x performance increase over the previous display-list based instancing implementation.

4.1.3.4 HUD Utilities

We implemented a collection of utility methods designed to facilitate rendering of Heads-Up-Display style components over a JView scene. Methods are available to temporarily switch to orthographic rendering and draw and fill various 2D shapes on the screen. Several methods provide integration with the Java2D Shape API to allow these classes to be used with JView.



Figure 16: HUD Text Element

Using the utility methods from HUDUtils, we added a background and optional outline to the HUDTextElement (Figure 16) to prevent text rendered on the HUD from becoming cluttered due to interference with other objects in the scene.

4.1.3.5 Histogram Grid

The histogram grid geometry has been enhanced to support grids that are rectilinear in any JView supported map projection. Previously, the grids had to be rectilinear in latitude/longitude coordinates. This allows histogram grid based visualizations to be applied to data that is gridded in arbitrary projections (Figure 17), as is common for weather data (e.g. GRIB files).

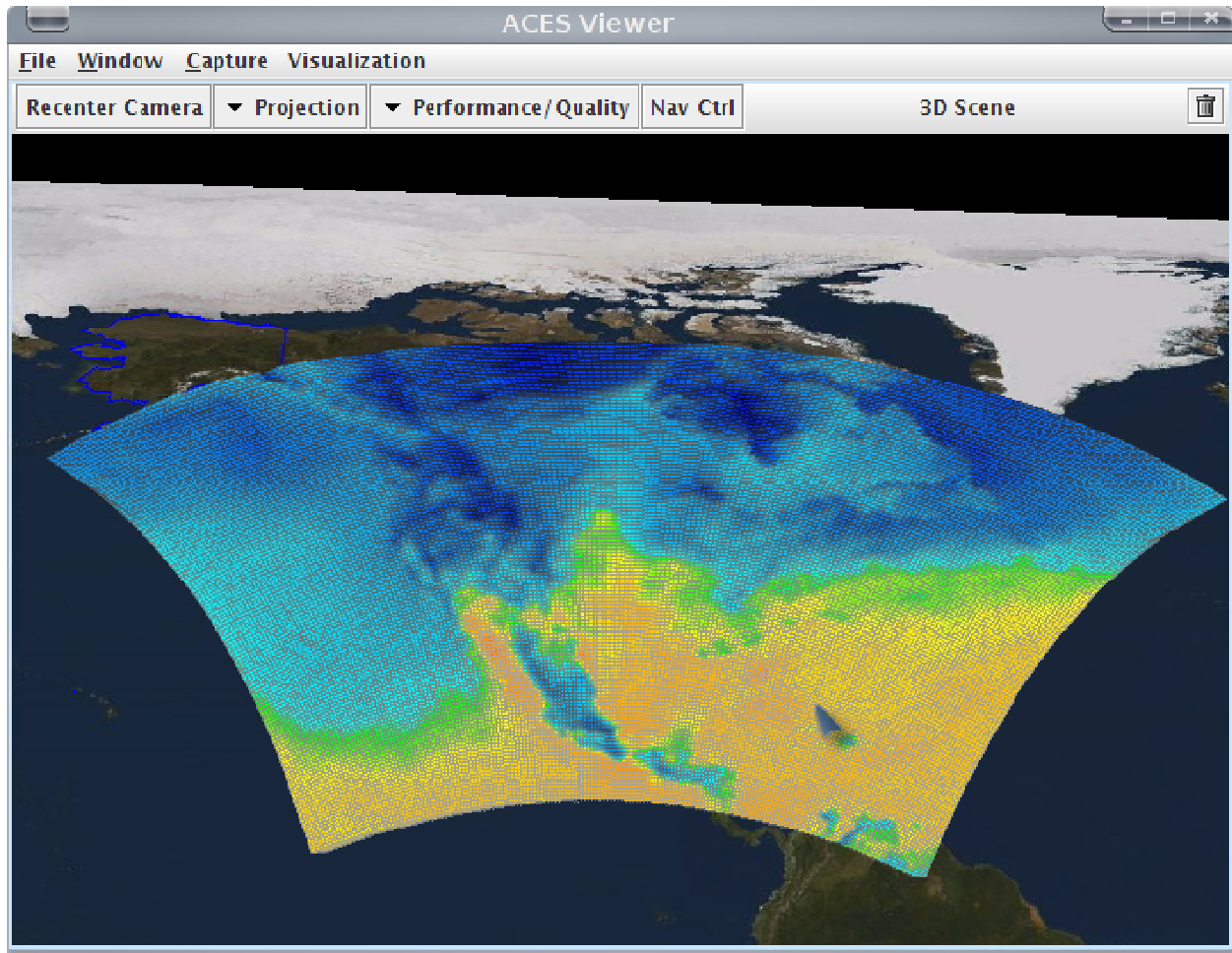


Figure 17: Histogram Grid displaying surface temperatures from a Lambert-projected data source

4.1.3.6 Sphere Segment Scene Element

The sphere segment is a shape that represents a portion of a sphere and there are six values that determine the portion. The min and max range that determines how the big the inner and outer surface should be based on the distance from the center of the sphere. The minimum and maximum azimuth values determine the vertical cuts of the sphere. The min and max elevation angles determine the horizontal cuts of the sphere. Figure 18 illustrates a demonstration of the sphere segment element in the DemoBrowser.

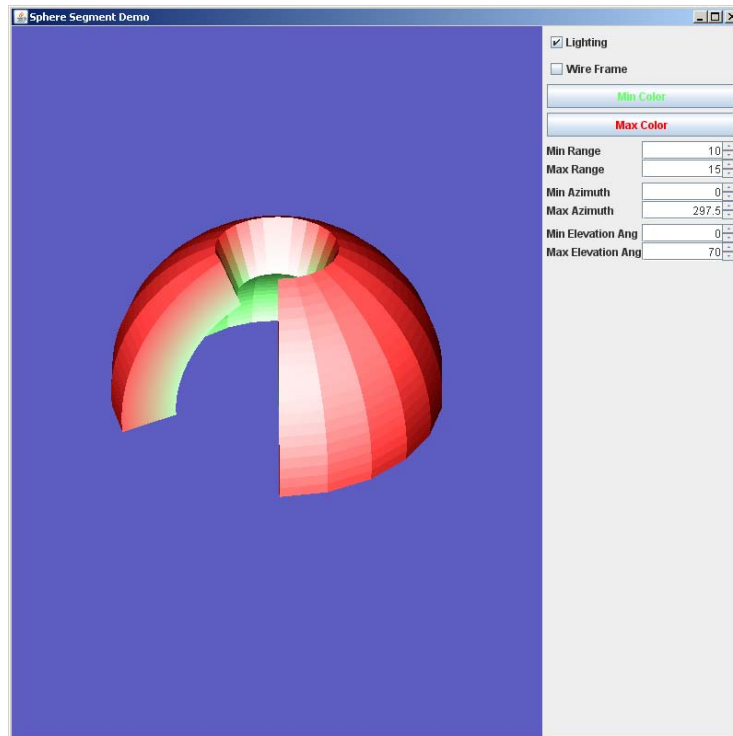


Figure 18: Sphere Segment Demo

4.1.3.7 Pin Renderer

Under the previous JView support contract, we developed a utility that can efficiently render many (thousands) of line segments (pins) that are intended to indicate pairs of related visual elements. The Label Renderer uses pins to show what point in space each label's text is associated with.

The Pin Renderer has been enhanced with support for drawing points at either end of the pin (Figure 19). The points are rendered using a symmetric 2D Gaussian texture map and OpenGL's point sprite primitive for extremely efficient rendering.

4.1.4.2 Keyhole Markup Language (KML)

Keyhole Markup Language (KML) is an XML file format that is used by Google® to represent geometry, points, and other geo-referenced information. During the previous JView support contract, CACI implemented a renderer for the KML based on a file parser provided by Google®. We wrote our own custom parser because the Google® provided version parser was slow (due to extensive use of Java's reflection utilities), and did not support the newest version of the KML specification. We also rewrote the KML rendering components to integrate with the new parser and to incorporate additional functionality.

4.1.4.2.1 KML Parser

Our KML parser uses standard Java runtime library systems to parse the XML file. The new parser does not need any third party libraries as the Google® provided parser did. Eliminating dependencies on third party libraries reduces the size of the JView download and can improve application startup performance. The parser goes through each XML node and parses them individually. The KML parser is structured similar to the KML document structure. Having this type of structure makes it easier to render and model. The performance of the new KML parser is significantly faster, and instead of minutes to parse a KML file we can now load a file in seconds.

4.1.4.2.2 KML Graphical User Interface (KML GUI)

JView's KML GUI consists of the components to represent KML in a panel. The KML GUI is separated into five different components. The main component is the KMLTree which extends the Java's JTree and defines its own tree renderer and editor. The tree uses the KMLTreeModel to define the structure of the KMLTree and the model keeps track of KMLTreeElements. KMLTreeElement manages all the actions and rendering between the tree and the parsed KML. It creates KMLTreeElements using the parsed KML and therefore defines the structure of the tree. Any actions on the element are implemented with the KMLTreeElementListener which can be applied to the KML tree. Figure 20 shows the structure of the KML renders.

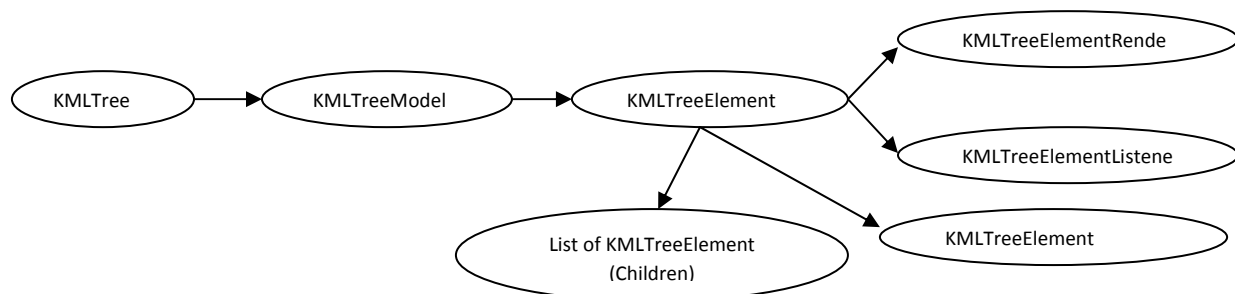


Figure 20: KML Structure

Since the KMLTreeElement actually controls the rendering, specific renderers can be applied for particular needs. For example with network links, it was important to show progress when connecting to an external link, and an intermediate progress bar pops up when connecting to an external source. Connecting to the web might take minutes to process so it is very important to give feedback to the user that the tree is in progress and processing. Figure 21 illustrates the network link renderer in the KML Tree and Figure 22 shows the KMLTree feature is available when the 3D element is clicked. Figure 23 shows the callout being displayed.

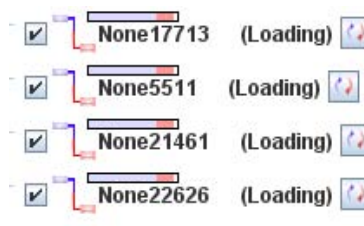


Figure 21: KML Tree Network Links

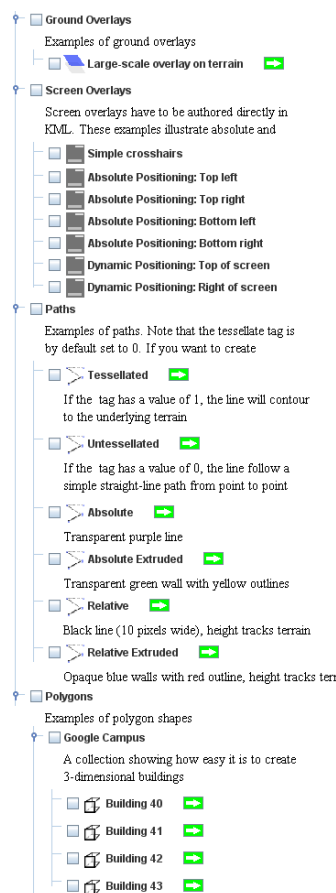


Figure 22: KML Tree

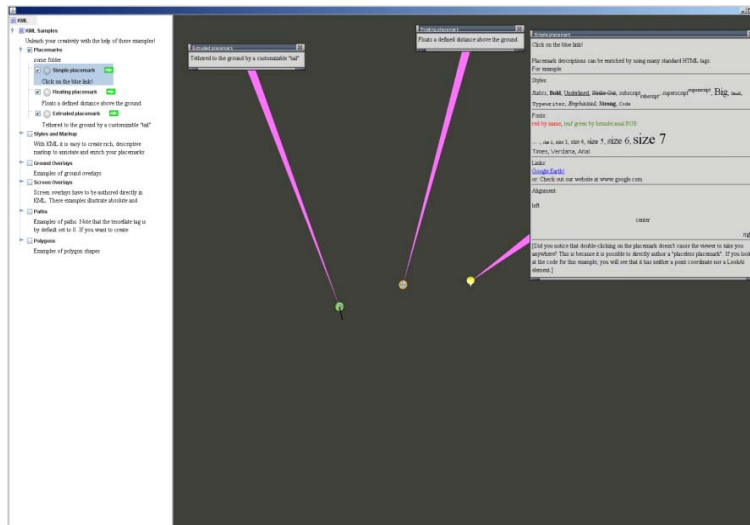


Figure 23: KML Callout

4.1.4.2.3 KML Placemark Point Scene Element

CACI created a new class called KML Placemark Point Scene Element to manage many KML point elements in a single JView scene very efficiently. The new class will show the placemarks icons, the name of the icon and if clicked on the icon a callout window showing the information of the KML point. Figure 24 shows the thousands of satellites being displayed on the world quickly with KML Callouts, and the KML Tree User Interface.

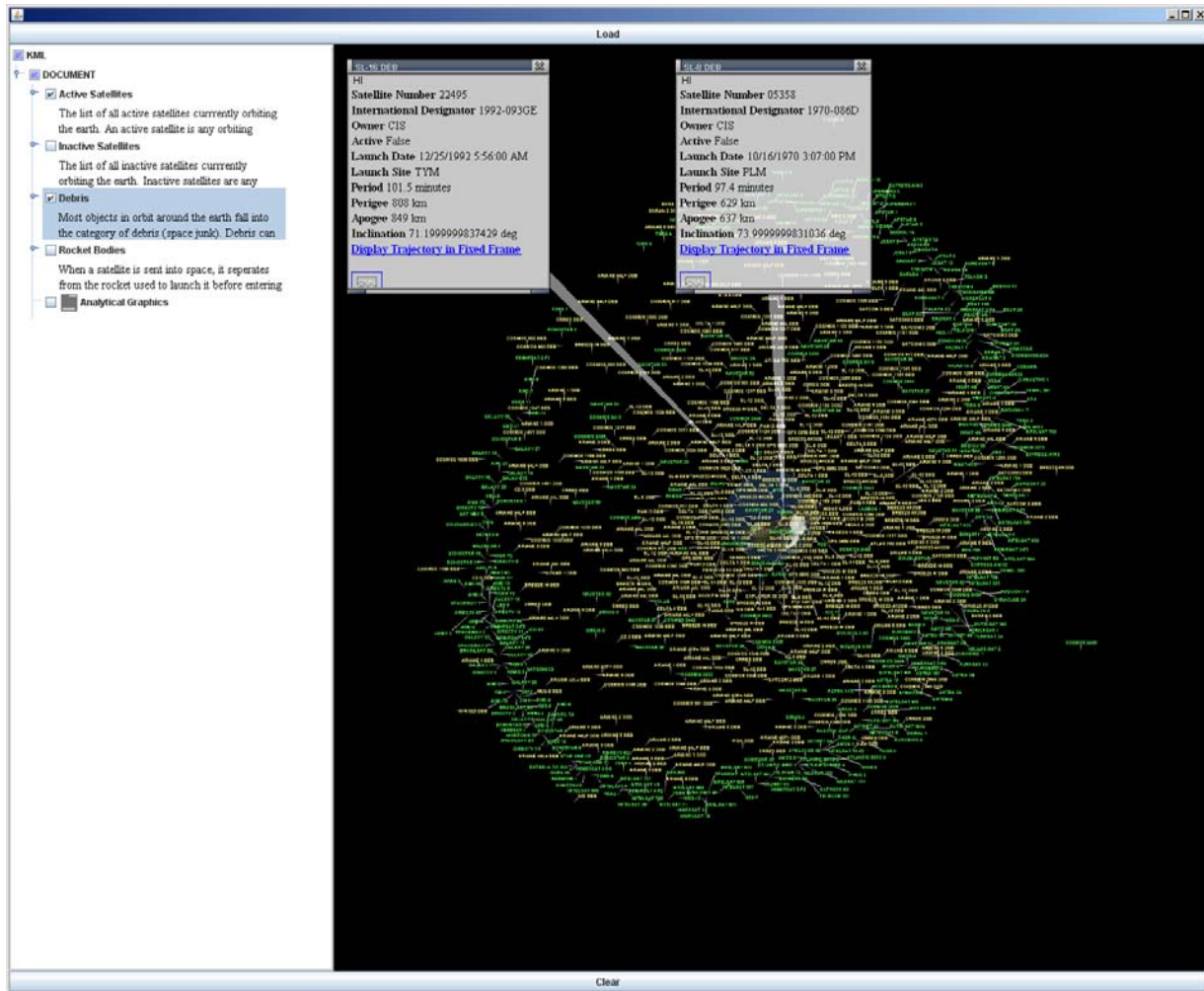


Figure 24: JView Scene with KML Placemark Point Scene Elements

4.1.4.3 Collada

Collada is a data format that was created to exchange digital artwork assets, such as 3D models, animations and kinematics data. This is implemented by having content creators adhere to an XML schema specifically designed to store 3D geometries, along with a host of other data that both statically and dynamically enhance those geometries.

4.1.4.3.1 Need A New Collada Implementation

Since 2008, Collada 1.4 has been supported by the JView model loader subsystem. The original Collada module used a pre-generated binary to facilitate the parsing of Collada (.dae) files. Some issues and limitations of JView's Collada file parser and renderer included slow parsing times, inability to effectively assimilate the new features of the ever-evolving Collada specification, and a relatively large binary file that must be packaged with JView to load models.

These problems stemmed from use of an auto-generated XML parser which facilitated rapid integration of Collada support into JView, however it often resulting in loading times on the magnitude of tens of seconds for a single model, usually much longer if textures are used.

Along with the aforementioned issues, the actual Collada files were not properly displayed within JView. Normals, matrices, and scaling were some of the issues that hindered the proper display of Collada models within JView. It was difficult to resolve these problems in the auto-generated parser, so we began implementing a custom solution.

The previous implementation also treated Collada as a secondary model to KML and KMZ. This was due to the fact that Collada were almost exclusively found associated with KML files or stored within KMZ files. However, as Collada has become more of a standalone model format, the current solution did not meet the needs of the growing model standard.

4.1.4.3.2 SAX Parser

To remedy some of the aforementioned issues, a Simple API for XML or SAX parser was written. The SAX parser performs a single pass of the XML file which allows the data to be put into memory where it is easily and quickly retrieved for the building of JView constructs. The previous system employed a random access read to the XML document to retrieve the information needed. This allowed retrieval of only what was needed; however, it included the additional overhead of relatively slow disk I/O.

After the SAX parser's single pass through the document, it generates an intermediate model which stores all data that is deemed relevant to building the 3D model using JView constructs. Due to the constant development on the Collada specification, along with the increasing needs of our customers, the SAX parser was designed to be easily extendable with minimal changes to the existing system.

4.1.4.3.3 What is Supported

Almost all available Collada models are using specification 1.4, and as of this writing, almost all models are supported to some degree. The Collada 1.4 specification handles many different aspects of representing graphical entities in a three dimensional environment. The Collada 1.4 specification's goal is to provide a median which can fully describe a three dimensional model, including how it looks (effects and materials), how to build it (geometry), and how it moves (animation).

Currently, any model using the primitive types of triangles, lines, triangle fans and triangle strips can be built and displayed using JView. We also support surface normals, lighting effects and texture mapping. Textures are only loaded into memory once in order to reduce the memory overhead. Translations, scaling and rotations are fully supported by the current implementation. Future graphical primitive types that are planned to be supported are quadrilaterals and quadrilateral strips. There are also some techniques which are not currently supported such as various shading techniques; however, the parameters required to allow these

techniques to work is already processed and stored for use by future enhancements to the renderer. Transparency is currently supported in a limited fashion, however will be more fully supported once shading techniques have been implemented. Figure 25 shows a sample Collada model loaded using the new JView model loader implementation.



Figure 25: Sample Collada model displayed using JView

4.1.4.3.4 Kinematics

Collada version 1.5 was introduced in August of 2008. The largest change that was made to the specification in version 1.5 was adding kinematics. The newest Collada version implemented a standard by which all non-scripted kinematics can be controlled. While Collada itself cannot specify how the kinematics-based animations should be managed, they provide axes by which subsections of the model can be transformed. Along with the axes, provided are the minimum and maximum ranges or degrees of freedom by which a part of the model can be rotated or translated. In Figure 26, a 1.5 model with kinematics describing the movement of the palm and the individual parts that comprise the fingers.

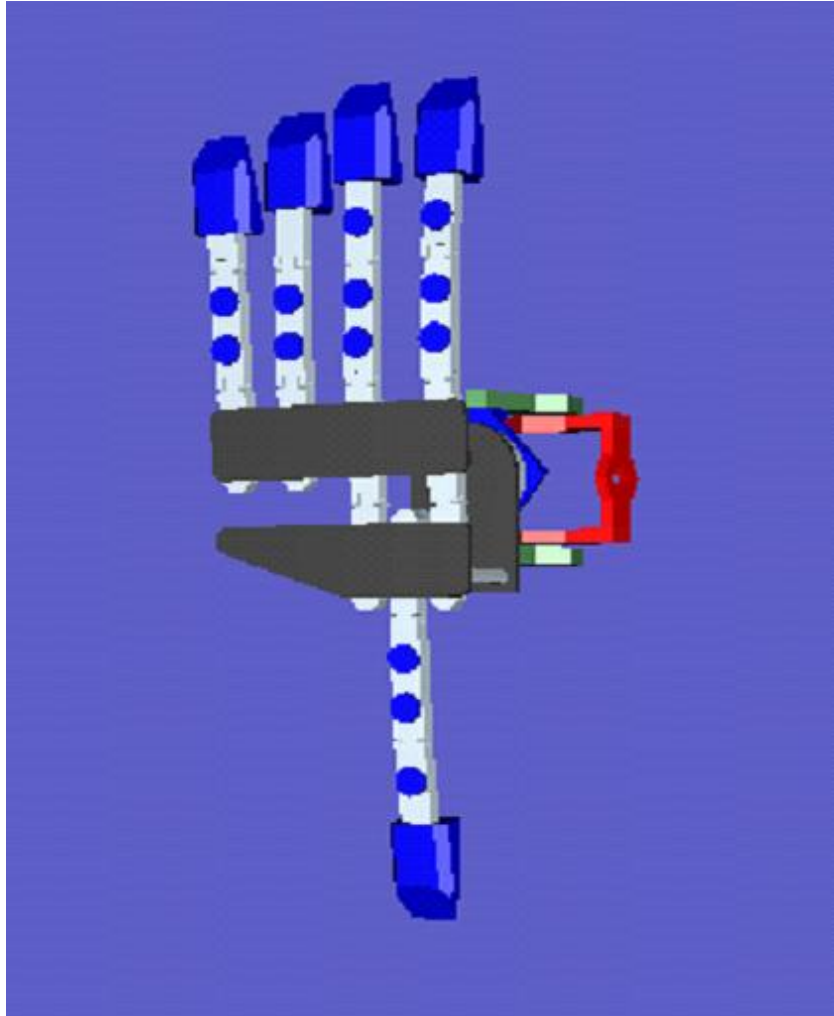


Figure 26: Collada Hand Model with Kinematics

Figure 27 shows the same hand model as Figure 26, with each joint rotated about 45 degrees counter-clockwise about the z axis. On the right is a simplistic control that was developed to showcase the possibilities of kinematics.

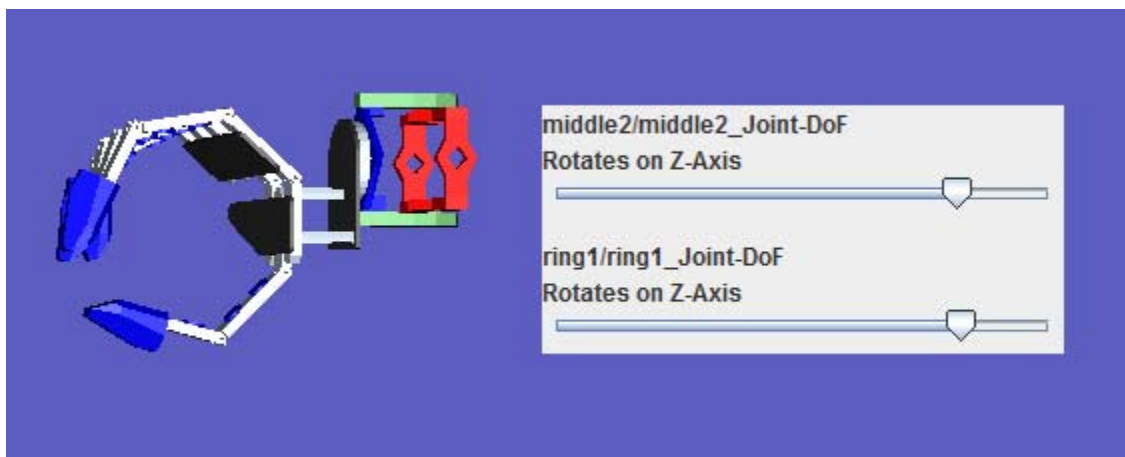


Figure 27: Articulated Hand Model

4.1.4.3.5 Phong Shading and Reflectance Model

The initial version of Collada renderer used the default pipeline shader to light models. The Collada specification allows a number of material models to be used, with the most commonly used ones being the Phong specular reflectance and Lambert diffuse reflectance. Per-pixel shading over flat shading is the main benefit of allowing these shading techniques over OpenGL's default Gouraud shader. The flat shading colors an entire shape, whether it is a triangle or quad, the same color. The improved visual aesthetics created by using the non-default shader does not cause a performance hit on the CPU as the work is offloaded to the graphics card. These two shading techniques are now supported in the latest version of the JView Collada loader.

The Phong reflectance model employs techniques to reflect more color back towards the camera from the light source. The amount of light reflected back is determined by the specular reflection value and the distance and direction the camera is from the light source. Where the light directly reflects back into the camera, the color seen will seem to be extremely bright, almost near white. The surfaces which have less light reflecting from it, the less shiny or bright it will appear. The outcome of the shading is areas of increased shininess which results in a more photorealistic model (Figure 28 and Figure 29). Support for Phong shading of Collada models was accomplished through implementation of several GLSL shader programs.

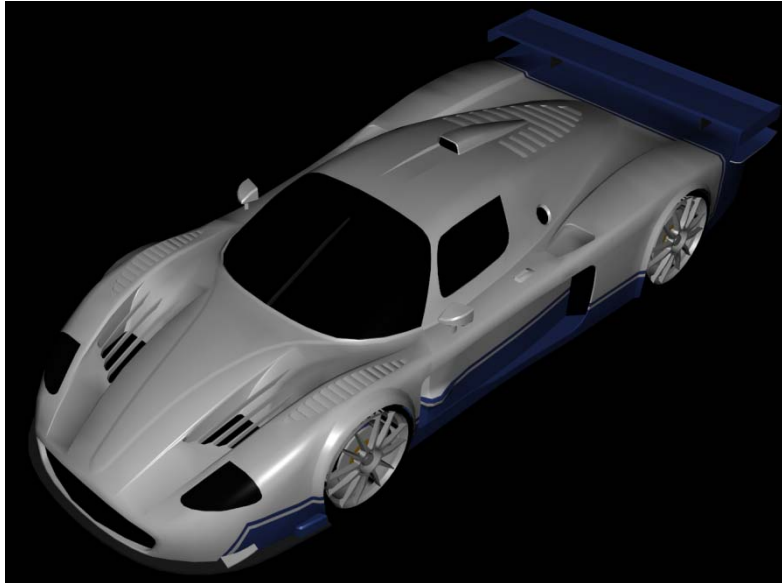


Figure 28: Phong shading and illumination applied to a Collada model.

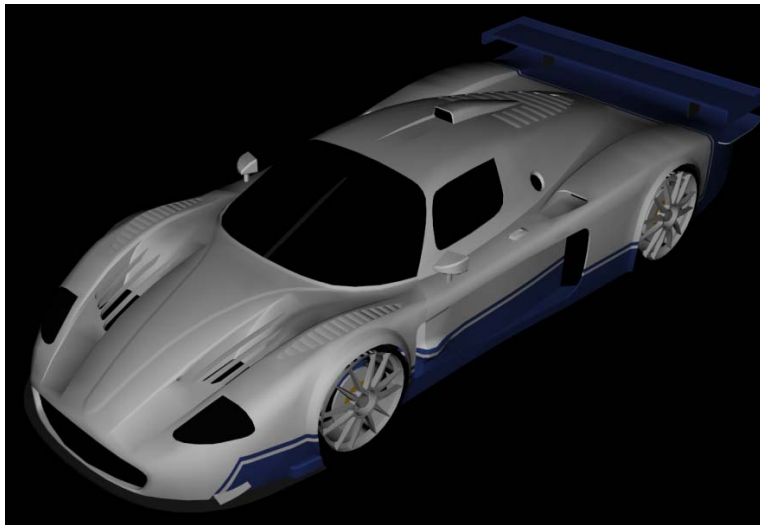


Figure 29: Lambert diffuse illumination applied to a Collada model

4.1.5 Release Management

4.1.5.1 Code Quality

We eliminated thousands of Java compiler warnings in JView. Many of the JView warnings are due to the lack of parameterized generic types. This type of warning was introduced by new features in Java 1.5 that did not exist in older versions of the Java compiler. Parameterizing Java Vectors and Lists helps in reducing runtime errors and helps to simplify tracing Java code. Other warnings were introduced during the normal development process.

We also developed standards for project source code layout and build system to be used across all JView 1.x based applications developed internally. This consistency ensures that a developer who is familiar with one project can easily navigate and build any of the other projects.

In order to improve the general quality of JView's code base, we incorporated a variety of static analysis tools into the build process. These tools identify potential problems in the code, including missing API documentation, concurrency issues, performance problems, and code style issues. We also added a number of unit tests to ensure that the code functions as expected when running, and to catch any regressions as new features are introduced. All of these tests can be run automatically during the build process. In order to encourage JView developers to contribute high quality code, and to fix any issues that the tools identify, we have also set up a continuous integration system that monitors the source code repository for changes, runs all of the testing tools when a change is detected, and generates a variety of reports that present any problems that were found. Using these tools, we have identified and fixed dozens of major problems, and hundreds of smaller issues in JView, and several JView-based applications.

4.1.5.2 Continuous Integration

To help improve and maintain the quality of the JView source code and architecture, we configured a continuous integration system that automatically builds the JView library from source code after each modification. During the build process, the source code is analyzed using a variety of static analysis tools, compiler warnings are recorded, and unit tests are automatically run. The continuous integration server generates a variety of reports and trends based on the collected information. It also assigns a build health status, and provides developer access to this information via a web server. The system provides similar services for all JView based applications developed under this effort. An example of the project health status is shown in Figure 30.

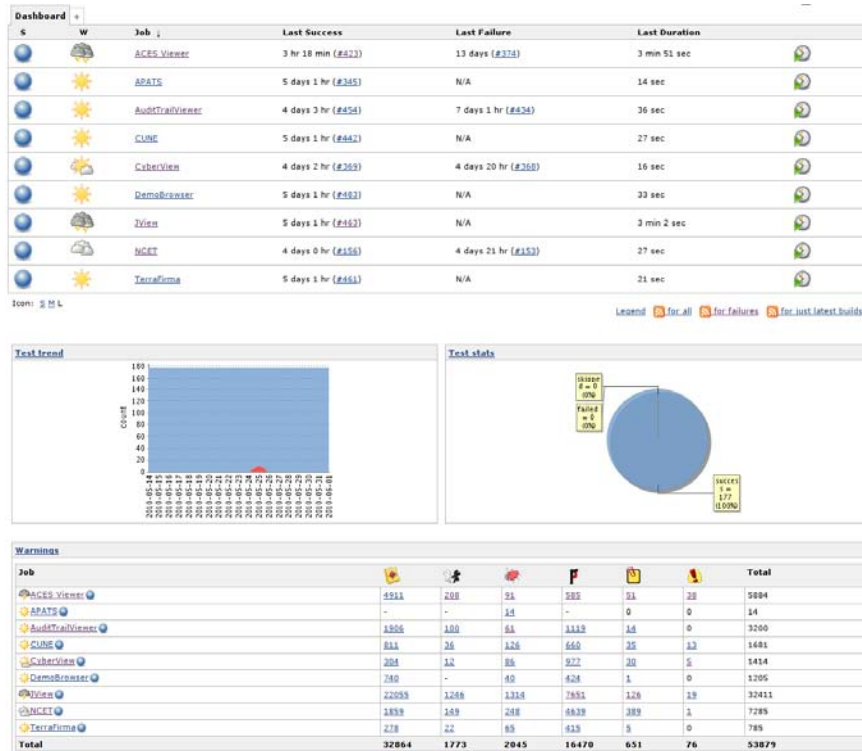


Figure 30: Project Health Summary in the Hudson Continuous Integration System

4.1.5.3 Application Installers

The latest version of Install Anywhere was purchased to create an automated installer for the JView, and other JView based applications. The previous Install Anywhere was not sufficient in our needs for creating installation media. It would not create installers with large data sets and would not work correctly with Java 1.6. There were significant changes between this version of Install Anywhere, and the earlier version. We spent some time learning and exploring this software in order to generate nicely polished installers for JView applications.

4.1.5.4 JView Releases

4.1.5.4.1 JView 1.5

We released JView 1.5, ACES Viewer 1.1, Cune Viewer 1.0 and the ATV 1.2 in August 2008. The release process involved working with a new installer generation application that allowed us to create native application installers for all supported platforms (Windows, Mac, Linux). Install Anywhere 2008 was used to create each installer. We encountered several problems with the installer software that we were able to work around with minor changes to the distributions.

After rigorous testing of the installers for each application on several platforms, we created icons for each installer based on images from the application. In Figure 31 the icons are listed with ATV, CuneViewer, JView, APATS and the DemoBrowser.



Figure 31: Icons

4.1.5.4.2 JView 1.6

The installer for JView 1.6 was created using Install Anywhere 2008, and versions were made for Windows, Mac OSX and Unix (x86 and x64). New versions of the ACES, CUNEViewer, ATV and APATS were created as well, based on the update JView libraries. Additionally, new icons and custom install graphics were created for the other applications.

Extensive testing was done on the installers (the creation of which ranged from five minutes to half an hour each). The most notable problem we found was with the Mac OSX installer, where a custom script had to be written to modify the application launchers, as the installer did not correctly wrap all the environmental variables needed for the programs to run. Additionally, the launchers for Linux cannot handle paths with spaces in their name. We found no workaround for this apart from warning the users to avoid installing to paths with spaces. This was determined to be a bug with Install Anywhere.

It was determined that this would be the final version to include *WorldDemo*. All of the functionality it provides has been subsumed by TerraFirma.

The features included in the installer are:

- JView 1.6
- DemoBrowser: A collection of sample applications used to demonstrate how to use JView for building applications.
- ModelThumbnailViewer: An application for viewing 3D models.
- TerraFirma: An application for viewing global imagery and terrain, similar to NASA *World Wind* and Google® Earth.
- *WorldDemo*: An application for viewing global imagery and terrain, similar to NASA *World Wind* and Google® Earth (replaced by TerraFirma).
- DTED Level 0: Terrain elevation data
- NASA Blue Marble: Imagery
- Geoname and political boundary database
- Documentation
- Java 1.6

4.1.5.5 JView Training

We provided a week-long development training session to researchers at the Human Effectiveness Directorate (AFRL/RH) at Wright-Patterson Air Force Base in support of several human factors experiments that they were conducting. The exchange also led to several improvements in JView with regard to usability, and support for newer stereoscopic display devices.

4.2 JView World

4.2.1 Rendering and API

4.2.1.1 Node Change Listener API

The Node Change Listener API provides a mechanism by which developers can monitor a (set of) location(s) on the world for elevation changes due to the level of detail geometry refinement changes as the camera moves through the scene. This API was introduced near the end of the previous JView support contract, and was substantially enhanced during this effort.

The initial implementation of the Node Change Listeners created a NodeChangeEvent Object every time a node change notification occurred. Early performance profiling determined this object would be created often and a NodeChangeEvent Pool was created to limit the affect of creating these objects repeatedly. During this period both the need for the NodeChangeEvent Object and the resulting object pool were removed. This was accomplished by using the QuadTreeNode as the key into the NodeChangeEvent HashSet, and then using the getValue method for the key to retrieve the original QuadTreeNode reference for which the event was triggered.

4.2.1.2 Camera Location Based Level of Detail (LoD)

In flight simulation applications, the camera is typically set for cockpit view. The view from the cockpit often leaves the World and looks off into space. Using the normal node area calculations for level of detail based geometry refinement would result in the World paging out all data, as the aircraft view returns to the World the data would then be paged back in. This results in very unfavorable visual effects, as well as possible jerking on less capable machines. In order to lessen this effect a new approach to determine node depth has been developed that uses distance from the camera to the node to determine the level of detail. This approach is not dependent on where the camera is looking, and will be set by where the camera is located. The previous node area based LoD metric is still used by default since this provides the best results in the most common usage scenarios.

Figure 32 shows an example of what is drawn using node area and frustum culling, the yellow pyramid represents the clipping planes of the camera, and it is clear that very little is drawn outside of the frustum. Figure 33 shows an example of the new camera distance mode with no frustum culling.

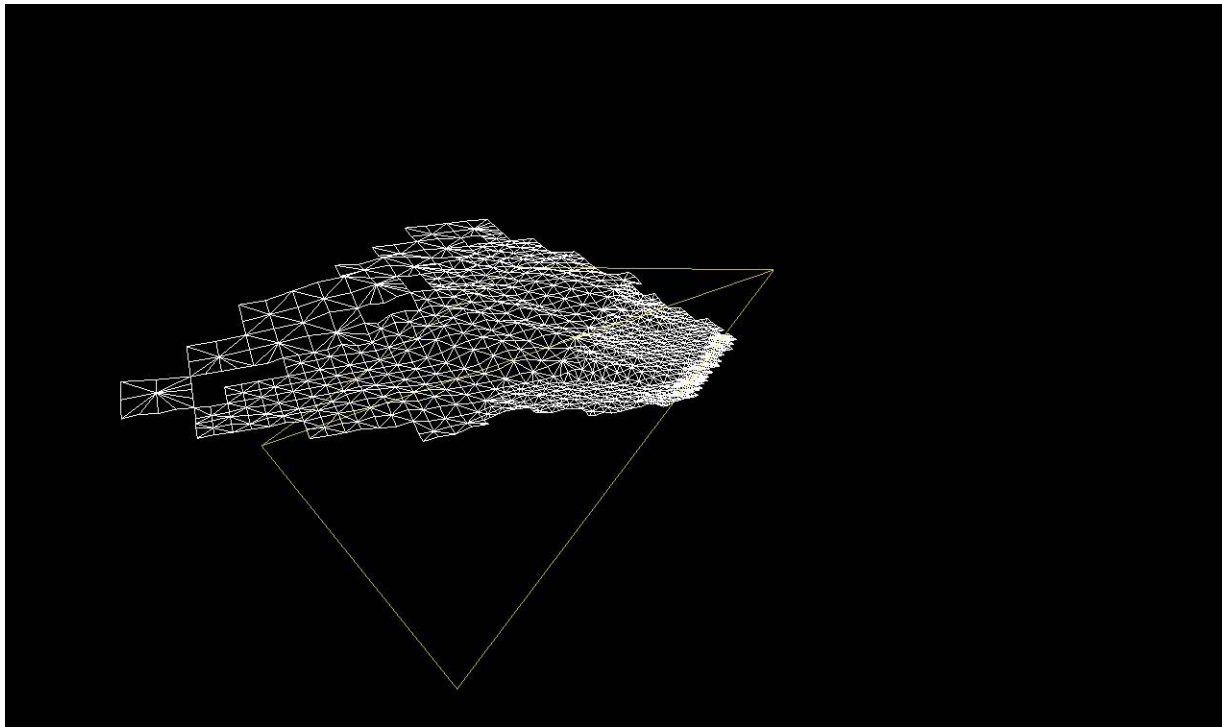


Figure 32: Node Area with Frustum Culling

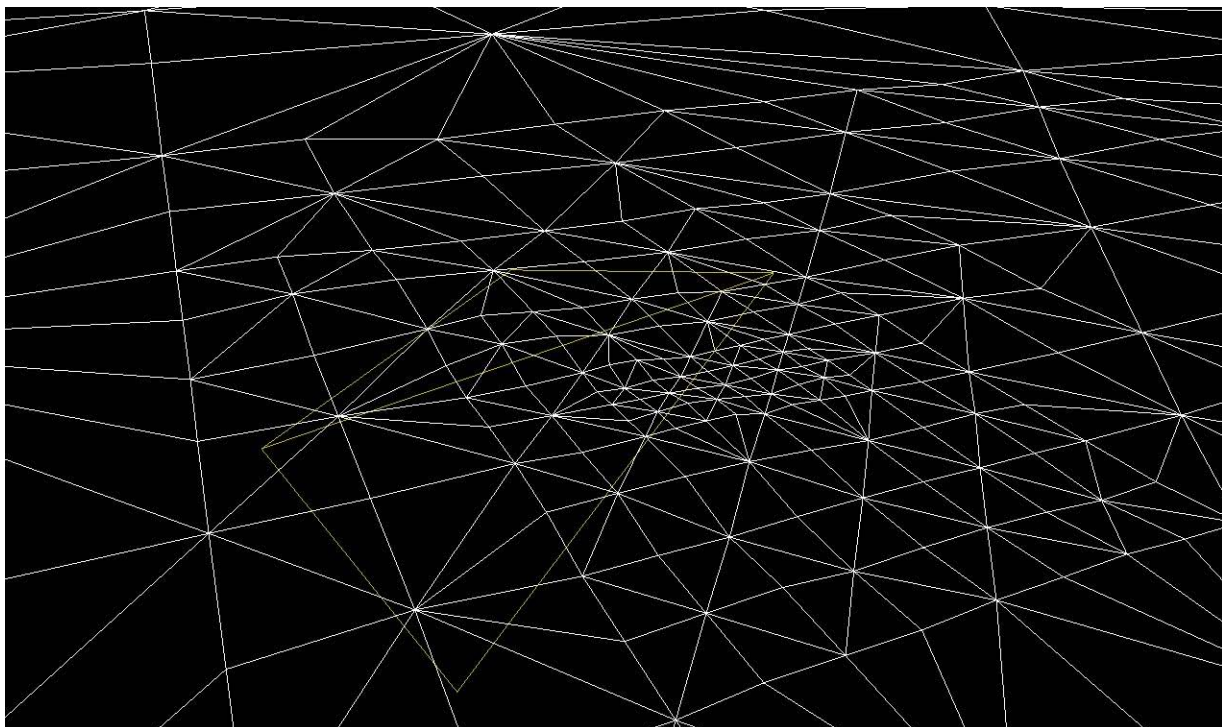


Figure 33: Camera Distance

4.2.1.3 Iterative Interpolated Geometry Refinement

As the LoD algorithm used by the world switches between discrete refinement levels, increasing or decreasing the resolution, a very noticeable pop occurs as the geometry changes. As an attempt to lessen this pop, CACI developed an approach that calculates how close to switching levels a node is, and then uses interpolation to limit the node vertices altitude from jumping large amounts all at once.

4.2.1.4 Elevation Interpolation

JView *World's* Elevation Queue object was modified to support several methods of interpolation on the default elevation data. Currently, allowing for DTED Level 2 in the absence of DTED Level 2 created a *Mayan Pyramid* effect. With the new interpolation methods, this creates a nice smoothed effect.

Interpolation methods:

- LOWER_LEFT_POST
- UPPER_LEFT_POST
- UPPER_RIGHT_POST
- LOWER_RIGHT_POST
- NEAREST_POST
- NEAREST_LL_UR_POST
- NEAREST_UL_LR_POST
- BILINEAR
- TRIANGULAR_LL_UR
- TRIANGULAR_UL_L

BILINEAR is now the default mode for JView World's Elevation Queue.

4.2.1.5 Vertex Buffer Objects

CACI began the task of converting JView *World* to use Vertex Buffer Objects (VBO's). The purpose of this change is to optimize rendering speed by limiting the amount of data passed to the video card each frame, and to prepare for compatibility with OpenGL 3.0.

The initial part of this change required adding logic to determine if the geometry of a node has changed. With a VBO, the data remains on the video card, and only needs to be updated when a geometry change has occurred. Logic was added that triggers a geometry update when a node splits, merges, or when a neighboring node changes its geometry.

We have successfully implemented a VBO based rendering system for the world that draws significantly faster than the previous implementation.

4.2.1.6 DTED Level 5

CACI began the process of supporting DTED Level 5 data in the World. Previously the World was limited to DTED Level 2. Since DTED Level 5 data only exists in limited areas and is classified, JView World will supplement with non-DTED data as it becomes available. The underlying post data structure will continue to fall on DTED Level 5 values, but when elevation values are loaded from non-DTED sources interpolation will be used to approximate DTED Level 5.

4.2.1.7 Mutable World Data Configuration

The initial version of the JView *World* object was only designed to have one view at a time. That version was enhanced to allow for multiple *World* views simultaneously and since the *World* is very capable of exhausting even a high-end computer's texture and main memory, data was shared between all *World* views, minimizing utilization of machine resources. JView *World* determines what directories to load its data from by a *WorldDataConfig* object passed to the *World* during construction. Previously, all *World* views would inherit this version of the *WorldDataConfig* object. This restricted all *World* views to that initial data object, and there was no mechanism to change it after *World* construction.

The JView *WorldManager* object was initially modified under a different project to allow users to modify the *WorldDataConfig* object after *World* construction. CACI completed this modification under this effort in support of the ACES Viewer capability to graphically configure this data object and to allow users to change it after construction.

4.2.1.8 Multi-Threaded Texture Fetching

The World now uses the JAVA *ThreadPoolExecutor* object to allow the Texture Loader to use multiple threads to simultaneously download imagery data. The default number of URL threads is 4, but may be changed by the user.

4.2.2 Imagery

4.2.2.1 GeoTIFF

GeoTIFF is an extension to the Tagged Image File Format (TIFF) that specifies a set of tags that provide geo-referencing information for an image. This format is commonly used for aerial and space based terrain imaging, and is additionally used by many industry standard tool sets. Many of the existing GeoTIFF images are extremely large (in terms of the number of pixels), and cannot be rendered in real time with reasonable memory and processing constraints.

We investigated new mechanisms for loading GeoTIFF data for use with the JView World. Specifically, we started with an existing GeoTIFF parser, and implemented a

subsampling interface using the Java Advanced Imaging ImageIO library, to dynamically subsample very high resolution imagery, for constructing dynamic reduced-resolution image sets from a single base image. Initial tests on this dynamic subsampling indicated that the first time ImageIO encounters a new image file, the subsampling routines as well as the overview image loading routines are extraordinarily expensive (on the order of 5-15 seconds). However, all subsequent access to that same image file (between reboots or extended periods of non-usage) are acceptably fast (on the order of 20-80 milliseconds). Effectively, this gives $\log_2(\text{IMAGE WIDTH} / \text{TILE SIZE})$ levels of reduced-resolution imagery from a single base. The reduced-resolution images are produced by creating a TextureSetDigitalGlobe with the correct "power" (0 gives the native 256x256 tile, 1 gives 4 native resolution tiles decimated to 1 256x256 tile, 2 maps 8 native tiles as 1 256x256 tile, etc., up to and including power 6 to map 16384x16384 images to a single 256x256 image).

4.2.2.2 Google Maps Imagery

The initial support for maps.google.com imagery in JView World was based on a lookup table for Mercator latitude values valid up to the 17th level of detail for satellite imagery. Since the initial support, Google® has increased its maximum satellite resolution level to 21. Level 17 is approximately .866 meter imagery, and level 21 is approximately .108 meter imagery. We generated a new lookup table incorporating all levels up to the 21st and Figure 34 shows a screen shot of a residential area in Albuquerque, NM with level 21 data.



Figure 34: Google Level 21 (~10.8 cm) Satellite Imagery

We modified TextureElementMercG to set browser information to make JView World look more like a request from the Mozilla web browser. Testing thus far seems to indicate this prevents a *lock* from Google®. Unfortunately, Google® has now implemented a 1000 file per day download limit. This is per server, and JView World rotates between all four servers. This results in JView World having a 4000 file per day limit from maps.google.com.

4.2.2.3 FalconView Imagery

JView users wishing to use JView World to display imagery intended for use in FalconView provided CACI with samples of their GEOTiff data. CACI examined the imagery and found that it had been through an orthorectification tool included with FalconView but had not been through a georectification process. The GEOTiffs contained incorrect location data and the orthorectification process on the original data was incorrect. CACI displayed the imagery as provided on JView World using the location data contained in each GEOTiff and Figure 35 shows an example of one of their data files on JView World. CACI notified the users that this data was flawed and needed to be regenerated from the source data using proper orthorectification techniques.



Figure 35: Falcon View Orthorectification

4.2.2.4 Layer Manager

CACI developed the layer manager to manage the set of imagery that should be displayed on the world. It is divided into two parts, one is a tree structure that is used to turn on or turn off textures (Figure 36). The other is an order layer manager that is used to order the TextureSet layers and specify which layers appear on top of other layers.



Figure 36: Tree Based Imagery Selector

The texture tree is used to add or remove TextureSets from the world. The texture tree takes a World instance and a reference to an XML file that defines the textures to be displayed and an associated texture organization structure for the GUI. The XML file contains a set of TextureLoaderFactory elements that specify the name of a class that is capable of loading the texture sets. Inside the TextureLoaderFactory XML element, categories and textures resources are defined. Each category may also contain nested categories, describing a tree-like structure that is used to build the GUI. A category could specify three types of information: name, type, and value. The name attribute defines the name that will be displayed on the table. The type and value attributes provide information that the loader needs for the texture set. For each TextureLoaderFactory class there is a method that takes map of type and values specifying what texture set to load. If a category does not contain type and value information, it will be skipped and just used as a visual representation of a category. Figure 36 shows the TextureSetTree and Figure 37 show a sample of what the XML file would look like for the tree.


```

<TextureSetLoader>

  <TextureLoaderFactory class='.textures.textureloader.CampRobertsTextureLoaderFactory'>

    <category name="Camp Roberts">

      <category name="A Category">

        <category type="year" value="2000" name="year 2000">

          <category type="level" value="0" name="Level 0 (15.625m)">

            <texture name="1_1" type="imagename" value="1_1" enabled='true'/>

            <texture name="1_2" type="imagename" value="1_2" enabled='true'/>

          </category>

        </category>

      </category>

    </category>

  </TextureLoaderFactory>

</TextureSetLoader>

```

Figure 37: TextureSetTree XML Config File Sample

The layer order manager is an auxiliary part of the layer manager that allows textures to be moved up or down the draw order. TextureSets are normally ordered by resolution so that the best resolution is on top and the lowest resolution is on the bottom. In general, this practice works for most cases but sometimes a user may want to change the visibility priority of a texture set. A good example of this is you have translucent images that should always be visible over the other textures. Simply allowing the World to determine the visibility ordering based on resolution might result in the images only being visible when you are viewing the World at a particular range of distances from the terrain. The layer order manager presents a list of textures that can be re-ordered by clicking and dragging on the move button. Figure 38 shows the layer order manager.

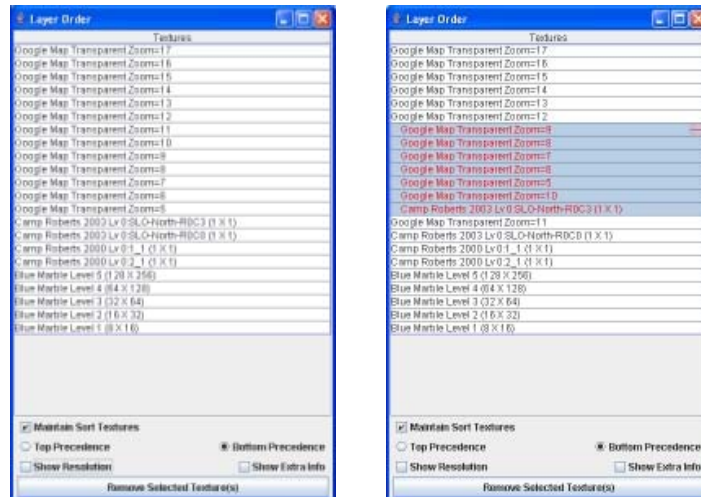


Figure 38: Order Layer Manager

4.2.2.5 Texture Masking

Several JView users have requested the ability to limit the World Element to only draw a small section of the Earth instead of everything seen by the camera. Their reasons have consistently been to limit the amount of data the users have to mentally process to a specified area of interest. CACI has considered several methods to achieve this including altering the World to only create limited areas. CACI applied the Textures function to mask areas not of interest and Figure 39 shows an example with such textures. In this example, an arbitrary roundish shape was used to demonstrate that any shape could be used. The area outside the region is 70% transparent black, 100% solid colors could be used to completely obscure the undesirable regions.

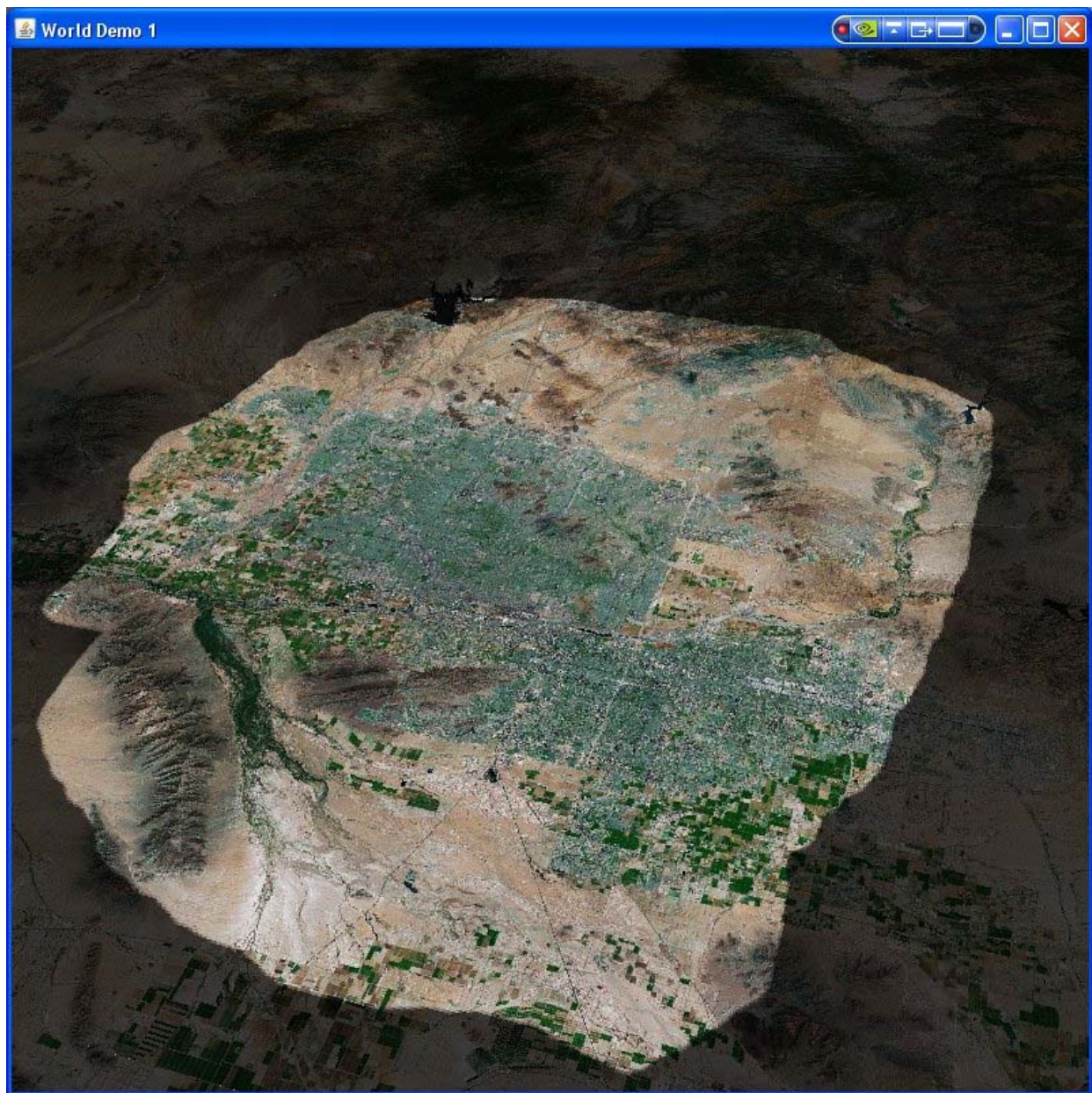


Figure 39: World with Texture Masking

4.2.2.6 Texture Union Optimization

As a texture memory limiting optimization in the first version of JView World, nodes that were completely covered by an individual texture did not load any additional lower resolution textures. When the node was completely covered by multiple textures from the same resolution set, there was previously no way to detect this, and at least one more layer of lower resolution imagery would be loaded. This was necessary since each node must have complete imagery coverage. CACI developed an algorithm to determine whether a node was completely covered by one, or multiple textures from the same set. This was accomplished with a complex

set of bounding boxes determining levels of coverage. This was significantly more complicated in the non-WGS84 coordinate system such as UTM/NAD83. Figure 40 shows a screen shot of a debug drawing of a node and all the bounding boxes used to determine texture coverage.

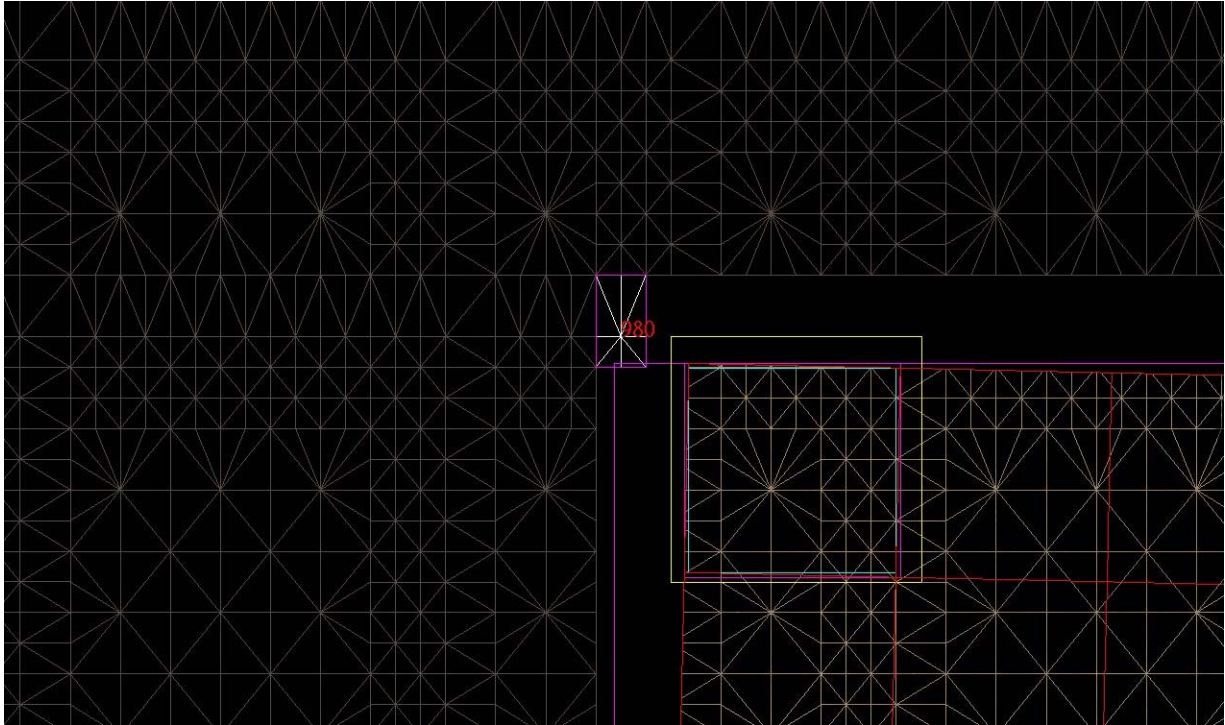


Figure 40: Bounding Boxes for NAD83 Texture Coverage of an Individual Node

CACI also added an option to load a user specified number of texture levels of lower resolution than are necessary. In this mode, the additional imagery is loaded into main memory but is not copied to the card. The purpose of this is to allow imagery to be "instantly" available when a user zooms out which results in the high resolution being paged out and the lower resolution imagery now waiting in main memory to be paged in. This results in a higher main memory usage, but does not result in a higher video memory usage on the graphics card. The greater the number of additional levels of imagery that are preloaded, the faster a user can zoom out without noticing undesirable "missing" imagery while waiting for lower resolution imagery to be available. This does not provide any gain when panning, or zooming in. When zooming in, the previously displayed imagery does not disappear when requesting higher resolution imagery, where as when zooming out once the user passes the threshold where it is inappropriate to display the higher resolution imagery it is immediately paged out. This can result in a few moments when only the coarsest Blue Marble imagery is available for display. Figure 41 shows the effect on the number of textures loaded over a test location before and after the union optimization, plus the effect of loading extra levels.

Setting	Number of Texture Loaded to Video Card
Before Texture Union Optimization	180
After	121
After + 1 preloaded level	162
After +2 preloaded levels	178
After +3 preloaded levels	180

Figure 41: Texture Count Before/After

4.2.2.7 Pixels Per Degree Threshold Optimization

Profiling the World element after JView 1.6 showed that the method that calculates QuadTreeNode texture coverage *initTextureDrawList* was one of the most expensive operations per frame. In order to reduce this load on the CPU, JView World was modified to only call this method when panning and zooming results in a threshold value being passed indicating that it necessary to recalculate texture coverage. Previously *initTextureDrawList* was called for all pan and zoom events for every QuadTreeNode. The threshold values are now calculated on the first initialization of the texture draw list, and only need to be recalculated if TextureSets are added or removed from the World. The pixelPerTexel ratio is still calculated each time a pan or zoom event occurs, but *initTextureDrawList* is only called when the thresholds are passed.

4.2.2.8 TextureSet API

The TextureSet API is the means by which imagery is incorporated into the JView World scene element. Previously all TextureSets required a reference to a World instance. This meant TextureSets could not be reused in multiple World objects. CACI modified the previously developed TextureSet API to allow individual TextureSets to be reused in multiple Worlds preventing repeated loading of identical imagery on the Video Card.

Previously the World object contained (Type) methods that allowed users to enable all the supported imagery types, but this was specific to a single World instance. After developing support for TextureSet sharing between multiple Worlds, these methods were removed from the World API, and were added to the individual TextureSet implementations. For example, the following syntax change would be typical for an application using two worlds enabling CIB data. The new way will, in future version of JView after 1.5, only load CIB imagery once, and then share between both Worlds. JView version 1.5 will still double load the imagery, but the new syntax will hide the future changes from the users.

Old:

```
firstWorld.initRPFTTextureSets(CIB, true);  
secondWorld.initRPFDTextureSet(CIB, true)
```

New:

```
TextureSetRPF[] theSets = TextureSetRPF.initSets(CIB);  
firstWorld.addTextureSets(theSets);  
secondWorld.addTextureSets(theSets);
```

4.2.2.9 RPF Catalog

JView World has had difficulty from the first version accessing RPF data sets near or over 1TB. On initialization, the RPF server has to search the entire file system for A.TOC files, the top most catalog describing the contents of each RPF data set originally distributed on CDs. This initialization can take several hours on large data sets, which makes it impractical to use in JView World. Up to this point most uses have only had data sets around 100GB. An RPF catalog option was previously developed that allowed the search results to be saved making subsequent access much faster, just as with the DTED catalog. The problem with this initial RPF catalog was that it used absolute paths to the data which meant it could only be used by multiple machines if all machines mounted the volume containing the data with the same name. For example machine one mounts data volume as drive “D:/”, and a second machine mounts the same volume as “/data” then the RPF catalog would not be able to store absolute path names that would satisfy both machines. CACI began the development of an RPF catalog that uses relative paths, and allows the addition of multiple catalogs in order to have a multiple machine, mount point independent option. JView World was modified to allow the users to configure this option through environment variables or through the XML configuration option.

4.2.2.10 Graticule Texture Set

Experimental tests were done to generate graticule lines using the TextureSetLine class. An algorithm was developed to generate line segments around the world based on starting angle and ending angles on a sphere (Figure 42). Multiple levels of texture sets could then be generated and added to the world, representing different levels of graticule detail. The world would then display only those texture sets that provide good coverage. Graticule lines that are too detailed or too convolved are culled out, and not rendered.

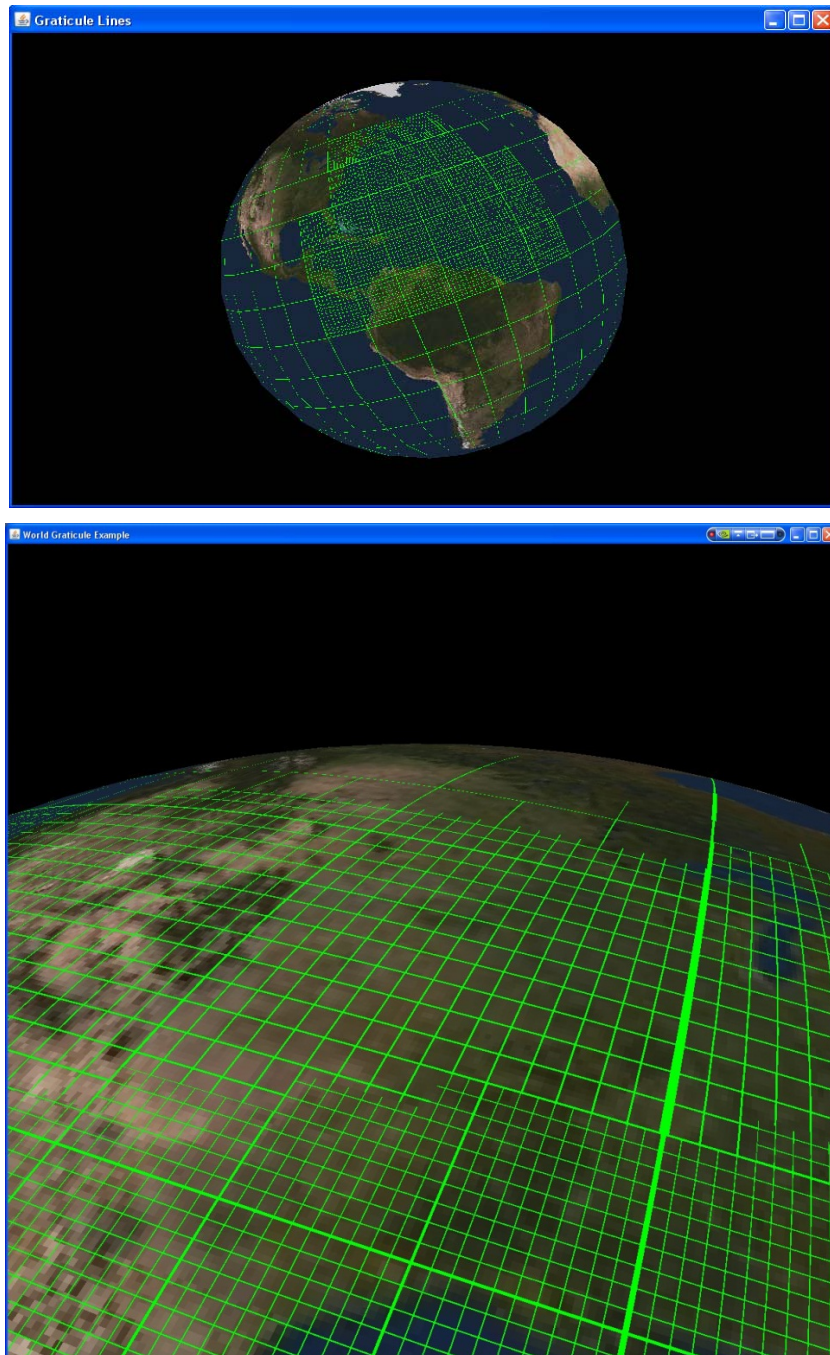


Figure 42: Graticule Texture Lines

4.2.3 World-Specific Scene Elements

4.2.3.1 World Raster Image Element

The `WorldRasterImageElement` is a scene element used to display icons or images onto the world. It takes an image, a `JView World` object, and location specifying where to place the

image. Due to the complex level of detail and view-dependant geometry refinement of the world, it had been difficult to attach visual objects to the ground in a scene. The WorldRasterImageElement handles this problem by listening to the geometry changes using a node change listener. Each node change listener is associated with a particular location on the world, and is notified whenever the geometry changes at the associated location. The WorldRasterImageElement's altitude is then adjusted accordingly. Another problem users have had when placing objects on the World is depth testing. Since the World is relatively large and graphics hardware has a limited amount of precision, the objects can be occluded by the terrain even when they should be visible. This issue manifests as flickering near the object as the camera is moved because sometimes the object is visible, and sometimes it is obscured. The WorldRasterImageElement overcomes this problem by providing options to disable depth testing or set the depth range which is optimal when a user wants to always see the icon and eliminate the depth fighting. Figure 43 shows a WorldRasterImageElement that is placed on the world at an altitude of 100 meters AGL.

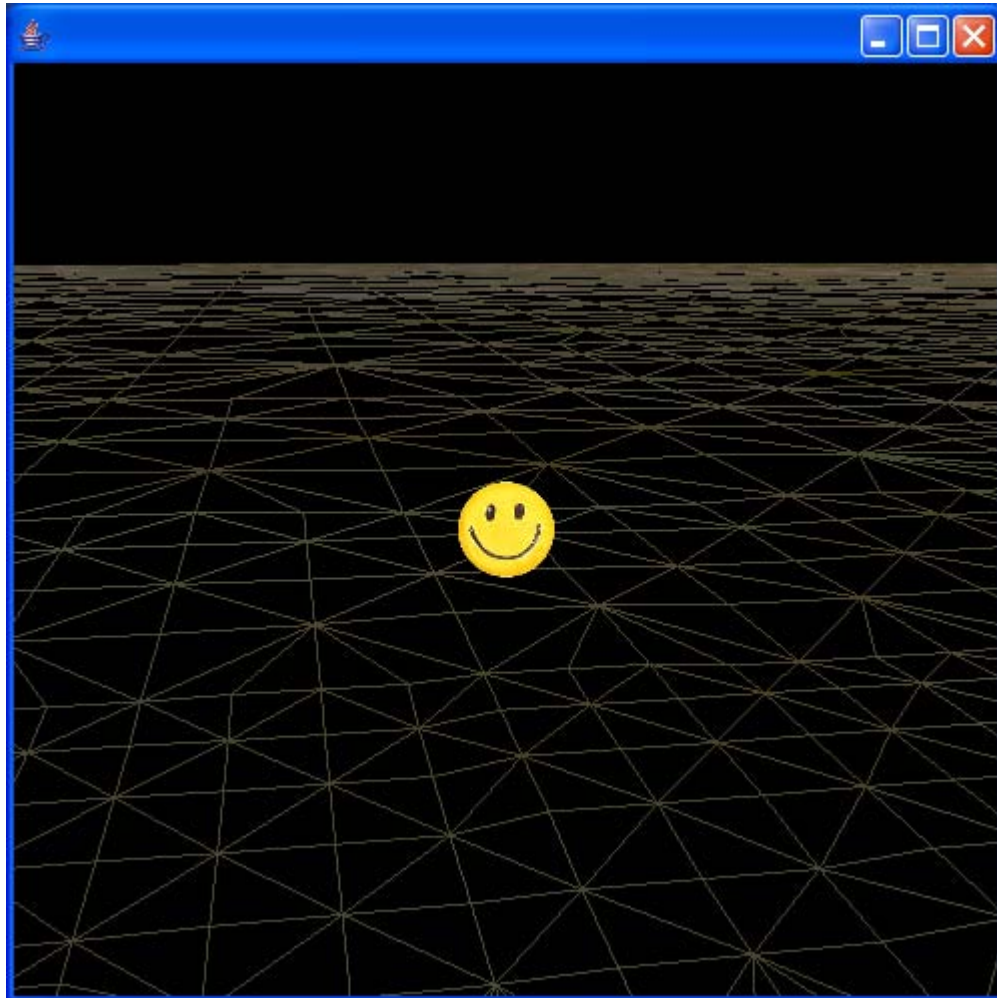


Figure 43: World Raster Image Element

4.2.3.2 Grounded Scene Elements

Grounded Scene Elements are elements that can be attached to the World's terrain geometry. They take the terrain elevations into account, as well as the view-dependant geometry refinement of the World element.

CACI developed a suite of grounding policies to model different behavior for a scene element attached to the World. Users can create a grounded scene element with an associated grounding policy and the element will adjust its altitude constantly to match the terrain elevations at the attachment point. Currently there are three different ways to ground an element to the earth, CenterToGround, ParellelToGround, and TouchGround. The center to ground places the origin of the model at the elevation of the ground. Parallel to ground is a clamping policy setting the origin of the model to the ground and orients the model so that the model is parallel to the ground triangle the origin fall on. The touch ground policy takes in a model and calculates the bounded verities, then with each vertex it will calculate the elevation of each vertex and places the model to the highest elevation so that the model is just touching the ground. Figure 44 shows the three grounded policies, parallel to ground is on the left, center to ground is in the center, and touch ground is to the right.



Figure 44: Grounded SceneElements

The grounded elements use the World's NodeListener API (developed at the end of the previous JView support contract) to monitor World geometry changes rather than calculating the terrain altitude with every render pass, yielding substantial performance gains.

4.2.3.3 World Info Element

We developed the World Info Element (Figure 45) to display the status of a World's texture and DTED queues, and some additional information regarding memory and graphics resource utilization on an in-scene heads-up-display style interface. This element was designed as a replacement for several Java Swing based components in the World Demo application that served the same purpose. It has the advantage of taking up less room in an application's user-interface, and can be configured to display additional information relevant to the application.

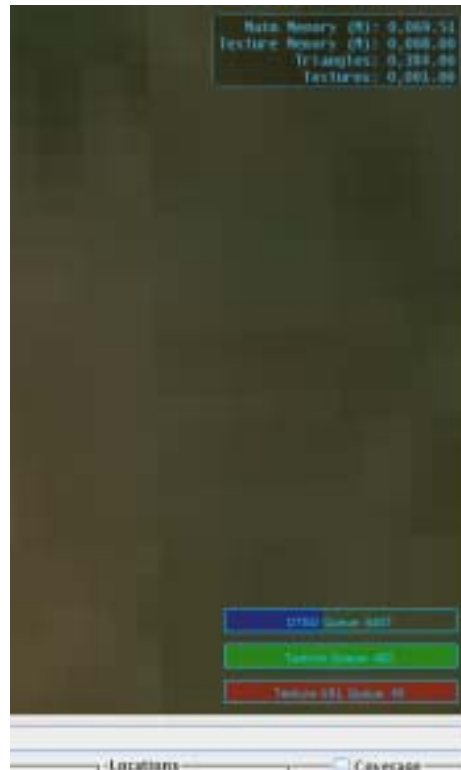


Figure 45: WorldInfoElement

4.2.4 Profiling and Debugging Tools

4.2.4.1 World Profiler

CACI developed the World Profiler to be a debugging tool that can be activated by anyone using the World. The World Profiler user interface has three sections that users can interact with to display various information about the associated World instance. After selecting what information should be displayed, the information window will query the World after each redraw to update the informational display. The profiler has a page history to view previous results and a logging option to save information to a file. The interface sections below the information panel are populated using Java's reflection API. Reflection allows us to identify and query the fields and methods in a particular class. The first panel is called the stats panel which shows a list of World statistics derived from the public fields in the WorldStats class. The display of these stats can be turned on and off by the user, which will update with window with the current set of active statistics. The second panel is the reports panel from which methods are called from the WorldReport class. These methods return a string that is printed to the information window. The third panel is the settings panel which can change the World settings. Figure 46 shows the World Profiler with some world stats and reports selected.

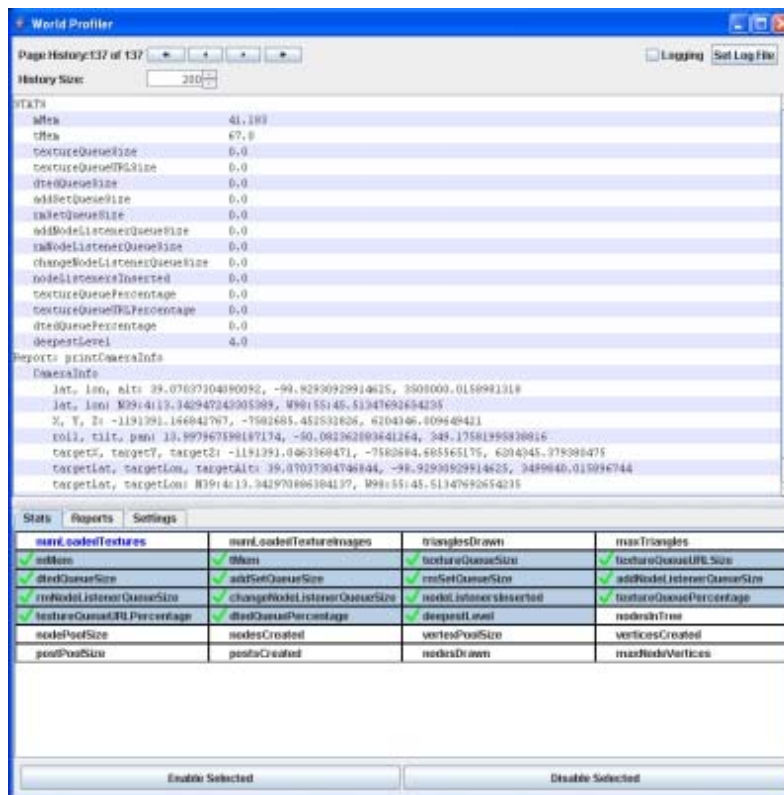


Figure 46: World Profiler

The World Profiler can be activated in three ways. First, it can be displayed as a standalone application and giving it a World instance to listen to. The second way is by calling the showProfiler method in the World itself. The last way and most useful way to showing the World Profiler is though a VM setting at the command prompt by adding `-DJVIEW_WORLD_SHOW_PROFILER`. Having the option to activate the profiler from the command line is very useful to users that are not familiar with the JView technology. It allows users to give us feedback without any complicated methods or code. The World Profiler is only displayed if a World instance has been initialized. An instance of the profiler will be created for each World that has been initialized.

The World Profiler reports section has an interesting graphical user interface interaction. The stats and report panel methods and fields are put into a JList where the user can double click to view the method or field in the information window. The reports panel is interesting because information needs to be retrieved from the user for parameters of a method. Normally this is accomplished by just adding a JTextField or JTable but adding a component to a JList is more complex. JList will not allow users to interact with the components inside the cells. In order to fix this problem, upon clicking a reports cell the contents of the cell are copied and displayed on top of the reports panel using a JLayeredPanel. Once the user clicks off the panel then the contents are then copied back to the original list. The copying of panels is a must because without copying, the components will be removed from the popup panel to the JList on repaint. By having this popup panel the JList are usable and feel more natural to use. Figure 47 shows the reports panel and Figure 48 shows the popup cell.

Stats	Reports	Settings
printLoadedTextureSets	✓ printLoadedTextures	printLoadedTextureImages
printNumberTrianglesHistogram	✓ watchTexture TextureName <input type="text" value="tile_5_0.png"/>	watchTextureSet TextureName <input type="text"/>
findNodesWithTexture TextureName <input type="text"/>	printNodeTextureList NodeNum <input type="text"/>	printNodeSetRangeList NodeNum <input type="text"/>
		✓ watchNode NodeNum <input type="text" value="3"/>
		printCameraInfo

Figure 47: World Profiler reports panel

✓ watchTexture
 TextureName

Figure 48: Reports Panel popup cell

4.2.4.2 Profiler Histograms

With the creation of the World Profiler, several new WorldReports were created to generate histograms for debugging and statistics generation. Figure 49 shows an example of the number of triangles drawn per node histogram. Each node will draw at least eight triangles, and will draw additional triangles when bordered by nodes at a lower level of detail.

```

Report: printNumberTrianglesHistogram
  triangles/node histogram
    triangles nodes
    0):
    1):
    2):
    3):
    4):
    5):
    6):
    7):
    8): 199
    9):
    10): 8
    11):
    12): 4
    -----
    Avg: 8.151658767772512
  
```

Figure 49: Triangles Per Node Histogram

Figure 50 shows an example of the number of textures drawn per node. Each node will draw at least one texture if `WorldSettings.useTextures` option is enabled.

```
Report: printTexturePassesHistogram
textures/node histogram
  textures nodes
  0):
  1): 182
  2): 28
  3):
  4): 1
  -----
Avg: 1.146919431279621
```

Figure 50: Textures Per Node Histogram

4.2.4.3 Texture Passes Histogram

For capturing additional statistics and performance metrics of the World a new *Texture Passes* histogram method was added. These histograms show how many draw passes were made for each geometry node. There will be one draw pass for each texture contained in the nodes texture list. Figure 51 shows an example of a typical World draw. In this example, 60 nodes made 1 draw pass, 128 nodes made 2 draw passes, and 106 nodes made 4 draw passes. In this example on average every node draws 2.5 times.

```
textures/node histogram
  textures nodes
  0):
  1): 60
  2): 128
  3):
  4): 106
  -----
Avg: 2.5170068027210886
```

Figure 51: Texture Passes Histogram

4.2.4.4 World Automated Test Program

The World Automated Test Program (ATP) is a framework and toolset developed to perform various quality, performance, and efficiency tests on the JView World scene element as it is developed. The tool tracks various statistics and expected behaviors over time to assist developers when regressions are introduced into the code base. Work began on this tool during the previous JView support contract, and continued to enhance it throughout this contract. This tool has helped find a number of difficult to isolate problems during development. One particularly difficult issue that manifested as incorrect elevation post values illustrates the utility of the automated test program.

4.2.4.4.1 Elevation Post Bug

For a long time JView World has had a timing bug related to the processing of elevation data that resulted in what was previously thought to be Digital Terrain Elevation Data (DTED) NULL values used by the World geometry in random places. DTED NULL is -32767 meters and indicates no data is available for a selected post in the DTED file. These values should be skipped and geometry refinement does not occur in areas that do not have DTED data. Previously this was very rarely seen. Since the bug was not reproducible it was difficult to do more than speculate as to what was happening. Recently, several new computers were purchased for the JView staff that all have high-end multi-core CPU's and graphics cards. On these computers this problem appeared in a very reproducible way. It was determined that the values are not DTED NULL but are sea level (0 MSL). This is only possible if post data is inserted into the World geometry list after entry into the elevation queue, and before processing of the post by the queue. This condition should not occur, as post data should not be inserted into the list until after it has been processed by the elevation queue. Figure 52 illustrates a close-up example of one of these invalid sea level posts being drawn over Denver, Colorado. The numbers shown in this illustration are post numbers used for debugging.

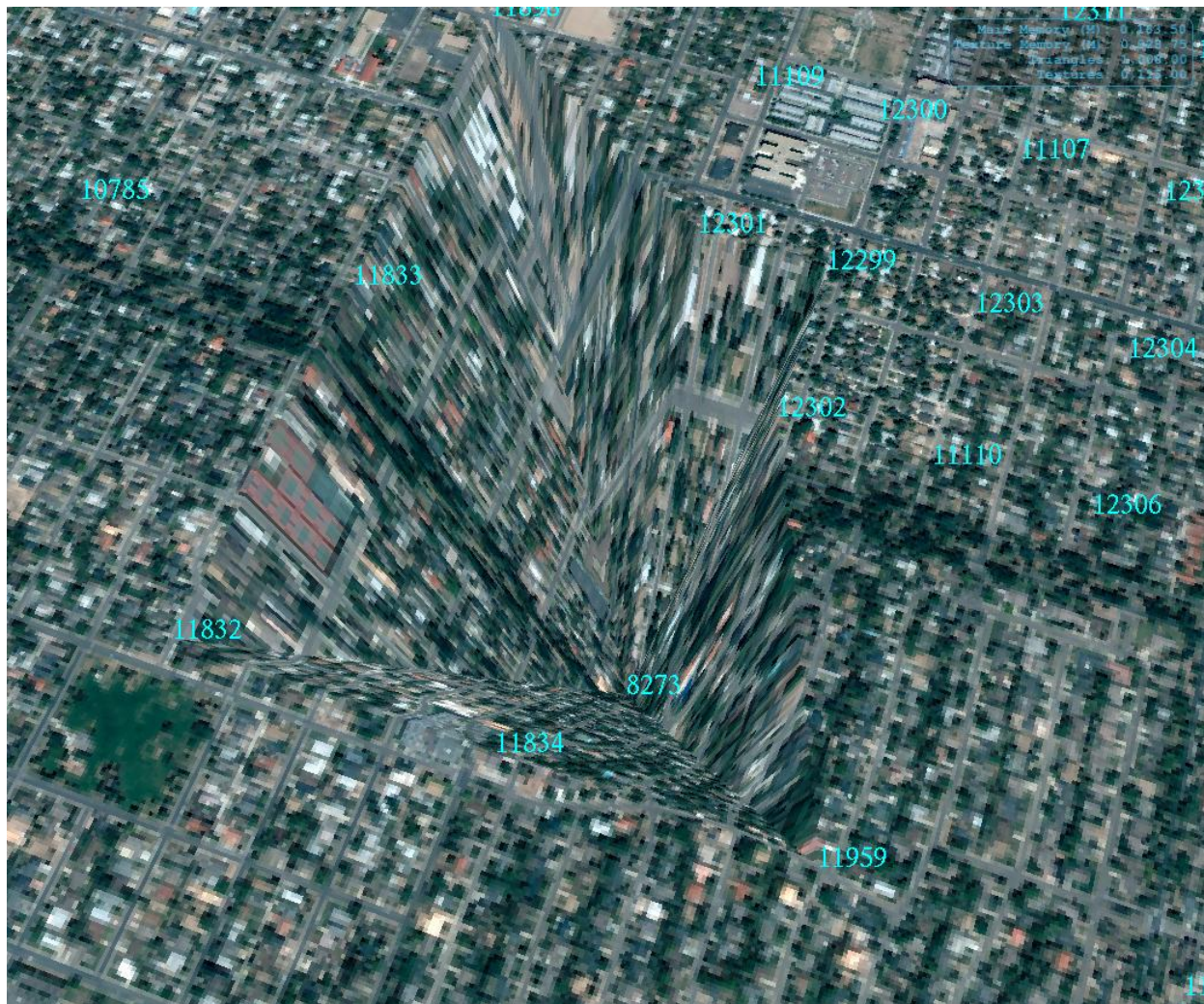


Figure 52: Invalid Sea Level Post

The ATP includes a series of tests that allow the JView team to determine if changes to the JView baseline and the JView World object result in any difference in data allocation and rendering. Using the ATP, two different problems appeared on the new JView desktop computers that were both found to be caused by the sea level post bug.

The first problem is only on multi World view test cases. The number of posts allocated from each World view is changing, yet the resulting image rendered is the same. The second problem is primarily a diamond shaped region of the screen that has an almost human unperceivable difference in pixels. This is immediately visible after a difference algorithm is applied to the screen shots. The difference algorithm was added to the WATP to help determine the cause of this problem. Figure 53 shows a screen shot of the differences with the pink pixels showing areas where differences occurred.

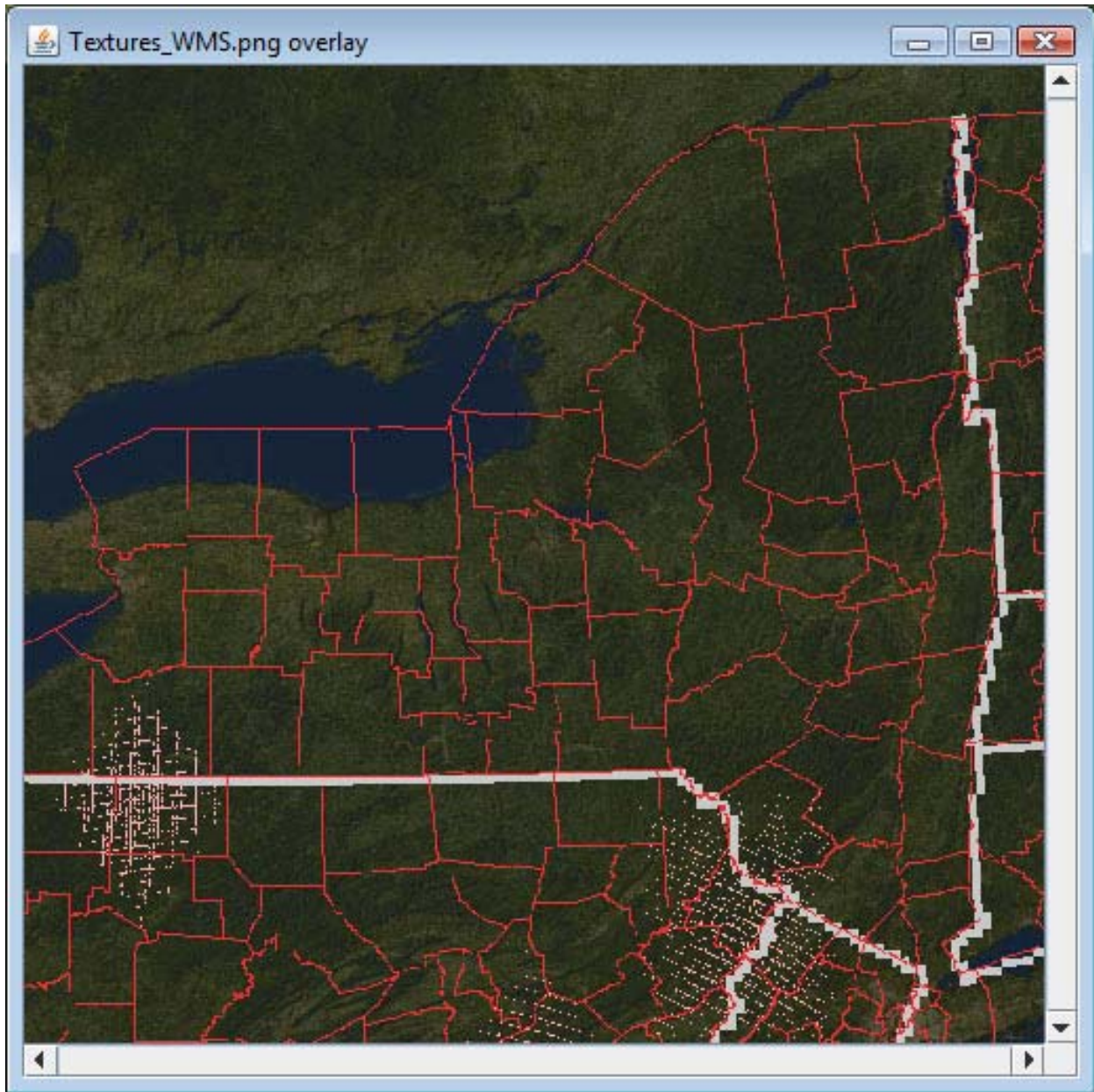


Figure 53: Diamond Shaped Difference in ATP

Graphics cards have been known to have differences of less than 20 pixels on these tests on other machines, but these differences are several hundred pixels. Since the diamond shape has a similar appearance to the problem with the sea level posts as shown in Figure 52, the assumption was made that these bugs were related. It was found to be a result of looking near straight down on the possible bad post over low altitude terrain, where as Figure 53 is at a much higher grazing angle over high altitude terrain resulting in a more obvious effect.

Two separate sources of these bugs were discovered that created this problem. The first was a incorrect if/else statement in QuadTreeNode that resulted in each post being projected

twice, once initially to 0 MSL, and a second time to the correct value determined by the DTED values in the elevation queue thread. The 0 MSL condition should only be reached in the case that useAltitude is set to false in the WorldSettings object. This is the location that the if/else statement was incorrect. This showed up on the faster machines because the elevation thread was actually processing faster than the render thread could return from loading the elevation queue, this resulted in the render thread overwriting the correct elevation with 0 MSL. The invalid altitude was inserted into the geometry since the elevation queue processing had actually finished at the point the 0 MSL projection occurred.

The second bug was each World in a multi World view application was reinitializing the DTED server, resulting in a period while the previous World views would be accessing the DTED server while another view was initializing. This results in a DTED server clamp value, which is defaulted to 0 MSL being returned. This problem was corrected by synchronizing all access to the DTED server and adding logic so that it was only initialized once. In both cases, the ATP proved invaluable and saved considerable time tracking down and resolving the issues.

4.2.5 Tools

JView users frequently request a subset of our imagery data for an area of interest. For data that was downloaded through URL's like Google® Mass and Terra Server these files are contained in a cache directory with naming conventions that match their original server names. Since the naming convention makes it difficult to redistribute a subset of this cached data during an earlier reporting period CACI developed a tool that assists with data extraction from our repository. The tool *RegionDownloader* parses a XML configuration file defining the area of interest and desired data and first attempts to copy that data from an existing cache, and will download from the internet if necessary. This tool also makes it possible to pre-download huge areas offline. The original version of this tool had repeated code and was not tied to JView. The original reason for this was to make it available to users that did not have JView and were downloading the imagery for use with other tools like FalconView. As JView and the World has evolved the *RegionDownloader* did not remain compatible with JView cache data naming conventions, and URL's changed due to Google® changing version numbers of their imagery. CACI modified the *RegionDownloader* to use JView TextureSets to calculate file coverage, naming conventions and URL values. This allows the *RegionDownloader* to keep in pace with JView, but does add the requirement to the tool that a user must have the jview.jar file in order to run the *RegionDownloader*. Currently the *RegionDownloader* supports Google® Maps, TerraServer, and NASA World Wind data.

4.3 Terra Firma

We developed a new JView based application to demonstrate several JView and JView World capabilities. This application will become a replacement for the World Demo application developed during the previous JView support contract. Terra Firma is designed as an application that JView developers can use as a starting point for their own projects.

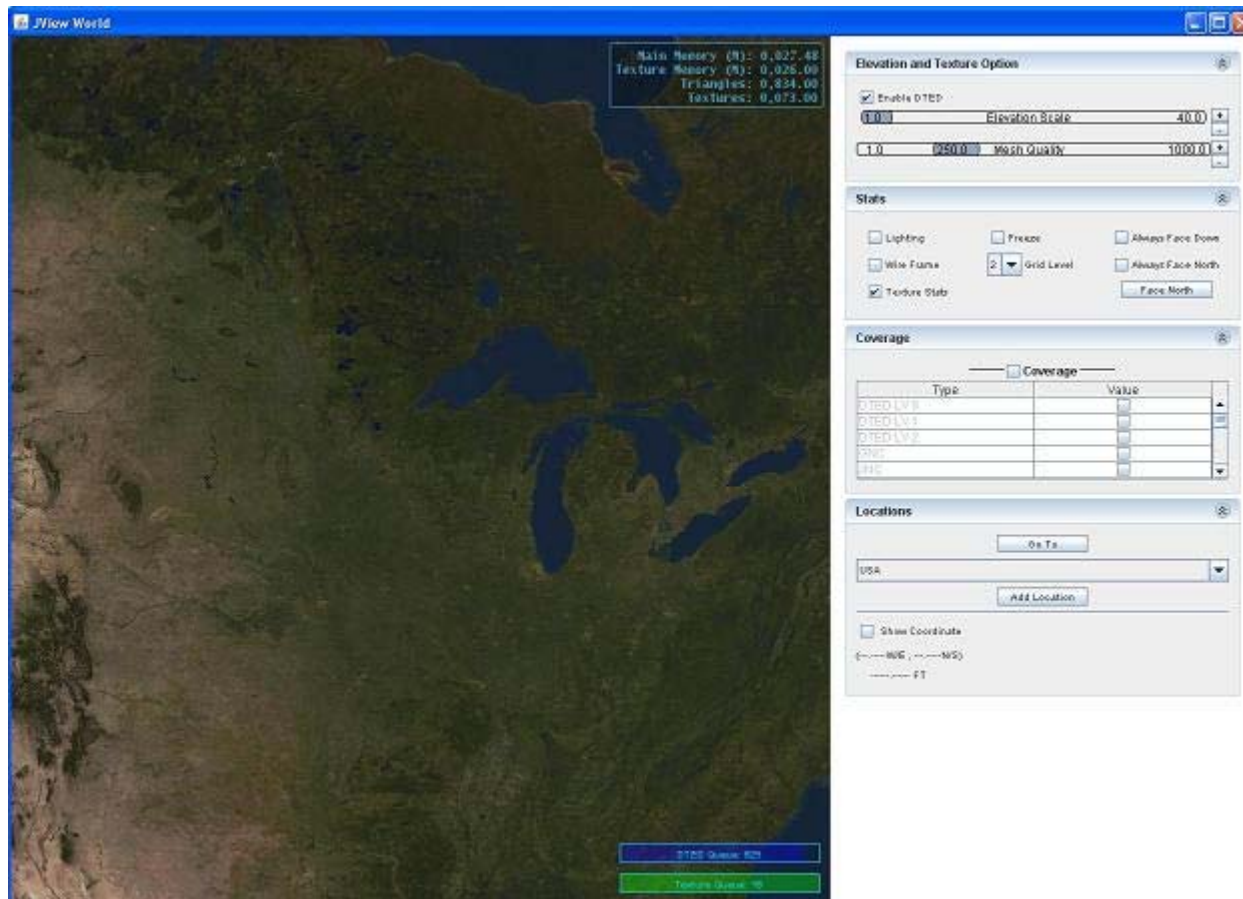


Figure 54: An Early Development Prototype of the Terra Firma Application

4.3.1 Terrain Profile Visualization

We implemented a 2D visualization of a terrain profile between two locations (Figure 55). This visualization relies on interpolated elevation data provided by a newly developed DTED interpolation utility.

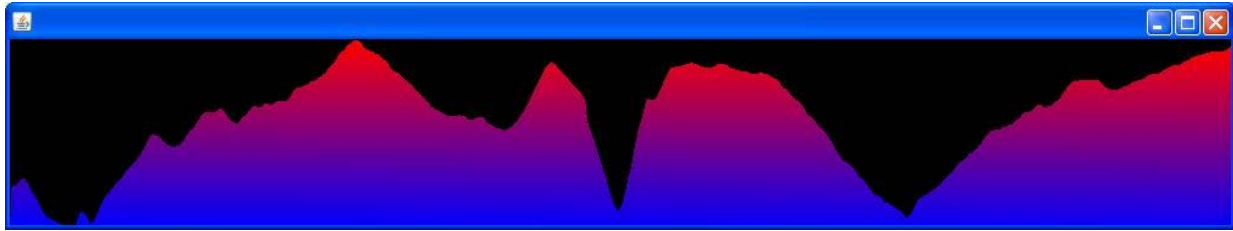


Figure 55: Terrain Profile Chart

Throughout the course of this effort, we incorporated several enhancements to the terrain profile visualization. We updated the terrain profile utility to handle absolute colors, via file containing colors and associated elevation values. The utility now returns the elevation values given the x position along the image, which is demonstrated in Figure 56 with the white line and associated elevation value. The utility will also return the latitude, longitude and elevation values, given an x value in the buffered image. This utility will also support setting the reference height and setting the maximum and minimums with reference lines as shown in Figure 56.

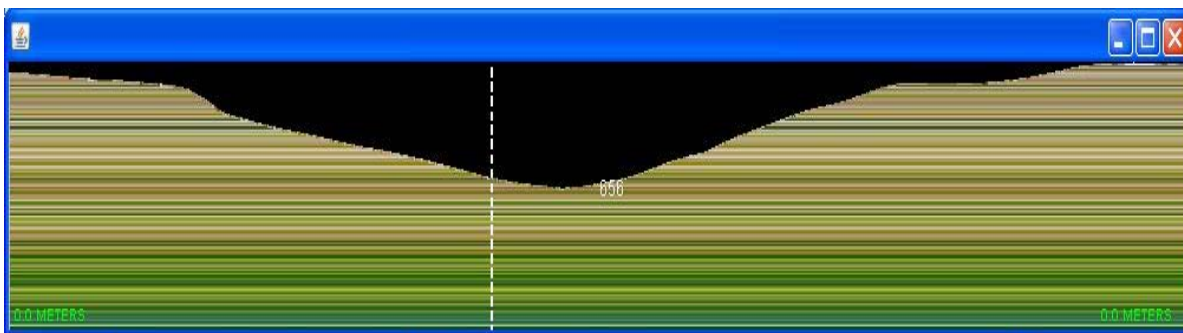


Figure 56: Terrain Profile

Figure 57 displays a Terrain Profile Callout, which uses the World Callout and Terrain Profile utilities to generate a transparent GUI in a JView 3D scene. The content area of the callout displays the elevations along a user defined path over some terrain. The Terrain Profile Callout This callout has indicators for the minimum and maximum elevation and an optional user defined elevation that is rendered as a horizontal line through the callout window. The callout is associated with a location in a 3D scene using a common attachment indicator (yellow wedge in the figure). The attachment indicator moves as the user manipulates the mouse cursor location so that it points to the corresponding location from the terrain path.

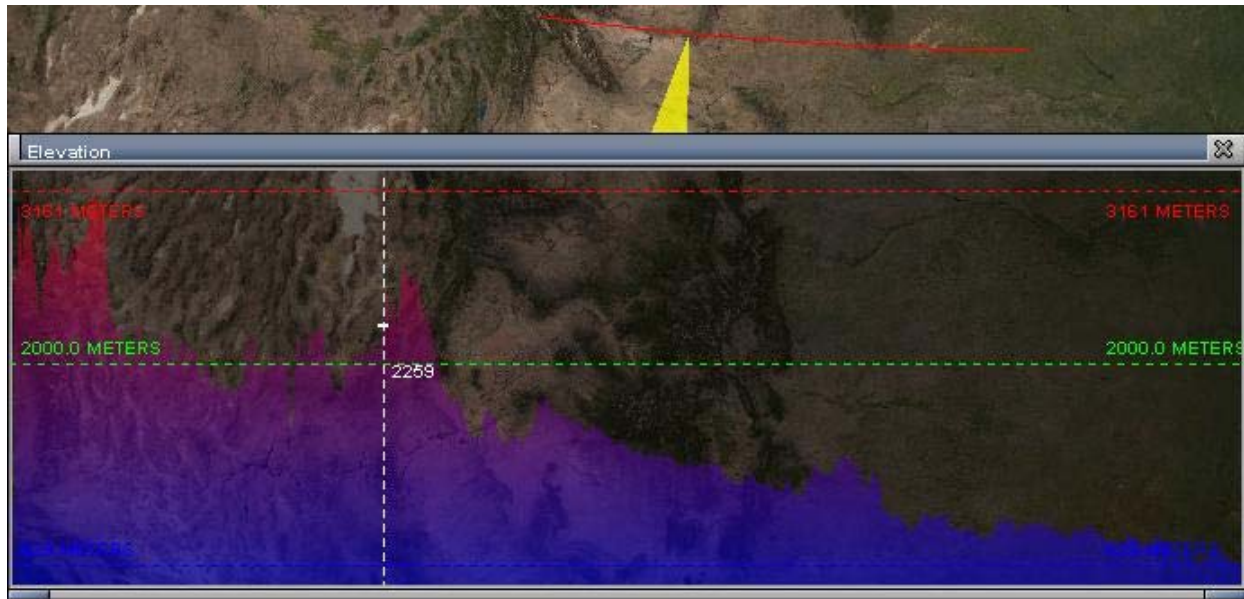


Figure 57: Terrain Profile Callout

The Terrain Profile user interface pulls elevation data from the DTED server in close to real time as the user is moving his/her mouse across the screen, as seen in Figure 58. The user can now see what the terrain will look like as the mouse is moved across the navigational chart in the World. The close to real time interactive elevation display is accomplished by using a small set of DTED points. Once the user finalizes the set of coordinates they wish to view in the elevation chart, a larger set of elevation points are pulled for a more detailed final display.



Figure 58: Elevation Projected Line

4.3.2 Geographic Locations Database

We created a GEO Names database using the free online “GeoNames” geographical data set, found at <http://www.geonames.org>, and the free geographic names found at <http://geonames.usgs.gov/>. The US features table in this database contains almost two million geographic names and locations in the US. A separate world table contains over 6.6 million names and locations for the entire world. A separate city table was created with 84,559 city data sets. The database has been incorporated into Terra Firma to allow users to selectively display feature names and locations over the JView World.

We used the embedded Derby database to store and access this data quickly and easily from within the Terra Firma application. Data retrieving optimizations were done by indexing the table data. Since information might be pulled using locations or coverage zones most of the data was indexed by latitude and longitude information. For small areas of coverage data retrieval and display happens under one second time. Additionally information was indexed based on country codes, city names and feature codes. Retrieval times for specific features such as city names, happens in the milli-seconds time frame. Before indexing, due to slow disk I/O time, the wait time for a specific feature could take as long as a minute. The indexing process in Derby uses a B+ Tree, which reduces the need to read much of the data and exponentially speeds up search times.

A geospatial search panel GUI was developed to help facilitate JView World users and developers. As shown in Figure 59, a mouse motion listener is used to draw an outline of a box in the world. This area is then used to search for features in the Geo Names database. The user has a selection of many different features to look for such as Dams, Buildings, Airports, Cities, etc.

During long-running database searches a wait/progress indicator callout pops up. The user has an option to stop the search thread if wait time is too long. The stop search mechanism will stop the search, but still show the data that was pulled from the database.

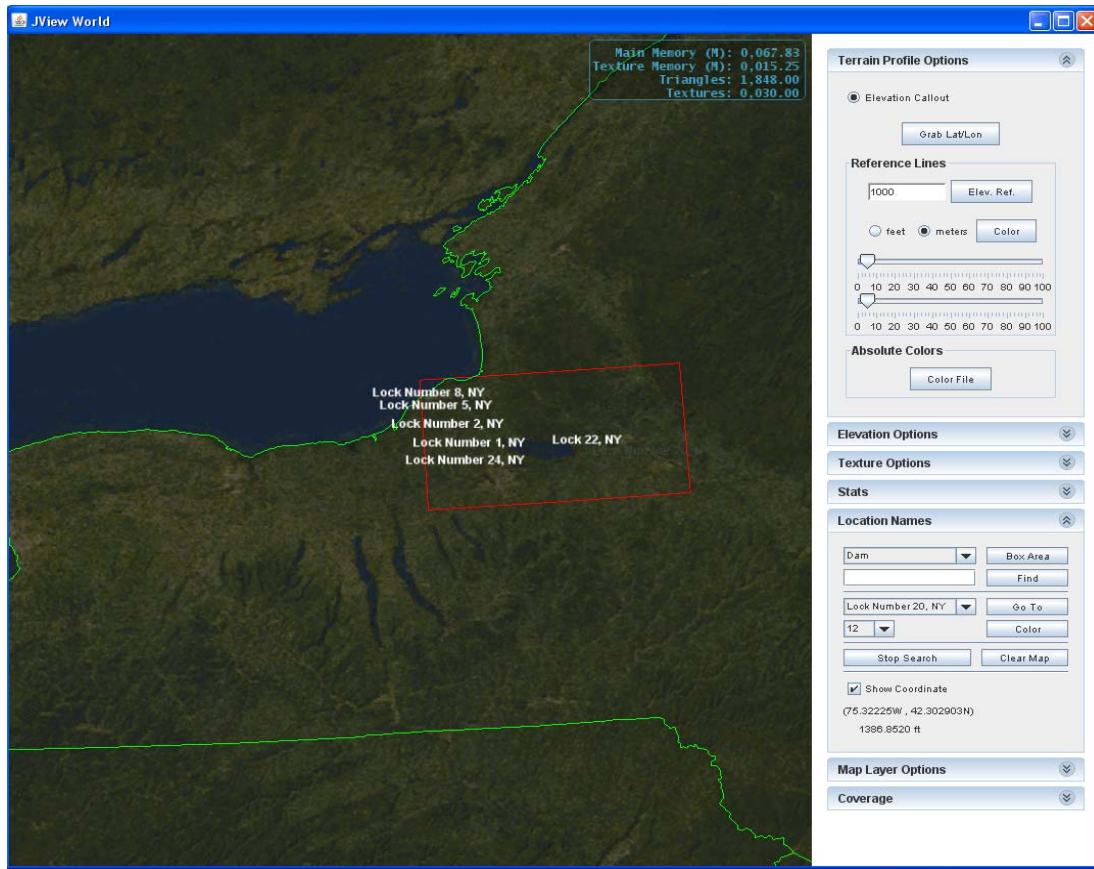


Figure 59: Search Area

We developed a clickable texture tree GUI, to help the user pick and choose which texture should be displayed in the World Viewer. Displayed in Figure 59 by default all textures are turned off, and the user adds additional textures to the world via mouse click. With this version of the GUI the world manages what texture layers should be drawn and no display ordering is done by the GUI.

City label names can be rendered in Terra Firma at a click of a button. As the texture area moves out of view the labels are dynamically removed and new city labels are injected. The city population size is taken into consideration when displaying city label names. Larger cities will generate larger labels and will come into view before smaller cities and towns (Figure 60).



Figure 60: Texture Set Labels

4.3.3 Geographic Search User Interface

Additional data from geonames.org and goenames.gov was pulled for country, country codes, US state, and state codes. These codes are now used to populate the new GUIs. The user now has more options for search criteria, such as the state to search in or the country shown in Figure 61 and Figure 62. The data set is now indexed using country codes, and state codes. These indexes could provide faster search times. The additional data also provides an increase in search flexibility. The new user interface also disables and enables GUI features based on the main types of features the user is looking for.

The three main search types available are World Cities, World Features, and US Features. When searching world cities or world features the country list is enabled. To decrease confusion when searching US features the US state code list is enabled and country list is disabled.

Location Names

All_World_Features Box Area

Airport US Feature

NY US State

Find

United States Country

United Arab Emirates

United Kingdom

United States

Uruguay

Uzbekistan

Vanuatu

Vatican

Venezuela

Go To

Color

Clear Map

Go To

Longitude Deci. Deg.

Elevation km

Figure 61: All World Features

Location Names

US_Feature Box Area

Airport US Feature

NY US State

NH

NJ

NM

NV

NY

OH

OK

OR

Find

Country

Go To

Color

Clear Map

Go To Location

Go To

Latitude Deci. Deg.

Longitude Deci. Deg.

Elevation km

Figure 62: US Features

4.3.4 WebStart

Java WebStart is a technology that allows Java applications to be launched from a web site without requiring permanent installation of the software. It can be used as a simple centralized application distribution mechanism to ensure users always have the latest version of the software. We experimented with a WebStart capable Terra Firma application to explore the possibility of distributing JView based applications in this manner. After some modifications to JView and Terra Firma, we were able to successfully demonstrate a WebStart-capable Terra Firma. We found the process to be relatively simple and highly effective.

4.3.5 Map Server

We configured the MapServer Web Map Service (WMS) server on a local system to evaluate its potential as a storage and distribution mechanism for various types of imagery supported by the world. We used the WMS Texture Set that we developed for JView World to access the server as an imagery source in Terra Firma. We found that this is a viable distribution mechanism, however it is far less efficient than reading the raw data from a local, or LAN based storage system. The performance was unacceptable on our test system; however with a more sophisticated image database and more powerful compute resources the performance may be improved.

4.3.6 Drag and Drop GeoTIFF

GeoTiff images can be dragged and dropped into the Terra Firma application and rendered based on the geo-spatial information in the image. The application will then use the TIFF's geospatial information to center on that spot in the world and zoom in to an altitude where the image is visible (Figure 63). The GeoTiffs are also added into the TextureSet image tree as options, which can be turned on and off.

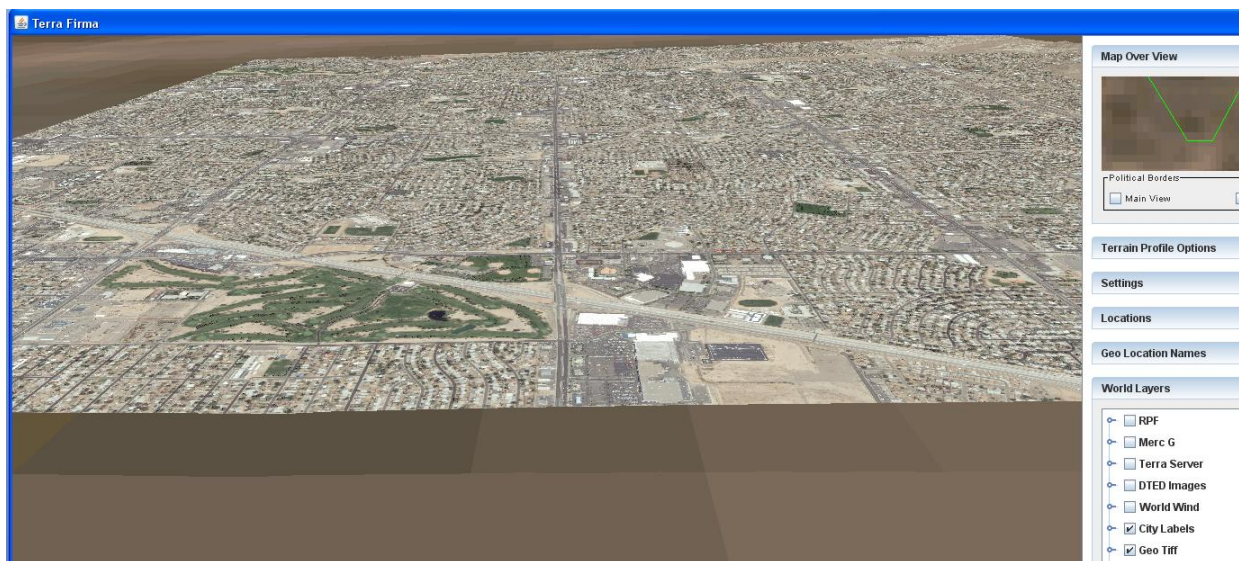


Figure 63: Dragged and Dropped GeoTIFF on JView World in Terra Firma

4.3.7 Overview Navigation

Overview Navigation Panel leverages JViews 3D camera frustum, projection calculations and the JView World Navigational features to provide a powerful navigational aid. JView World provides very sophisticated algorithms and programmer interfaces that provide useful 3D user interaction within the 3D geospatial environment. The World Overview display shows the bigger picture of the world with an estimated outline view of the world that is seen in the main display, shown in Figure 64 and Figure 65. The camera frustum is taken from the main world view. The frustum is the visible area seen in the World. The coordinates from that frustum are then used to draw a rectangle in the overview to show what part of the world is currently being viewed. The user has the capability to click in the Overview panel and the main view will move to that part of the world.

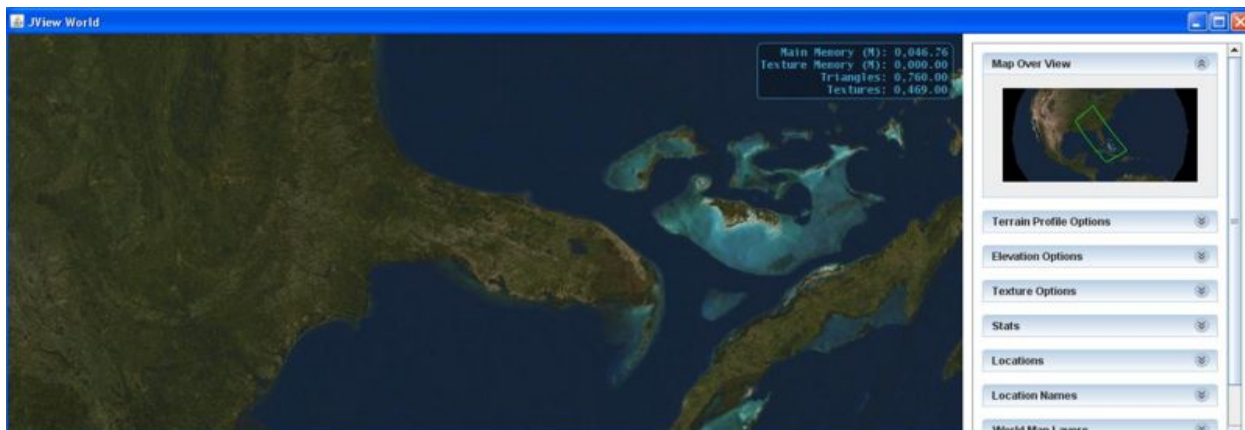


Figure 64: Overview

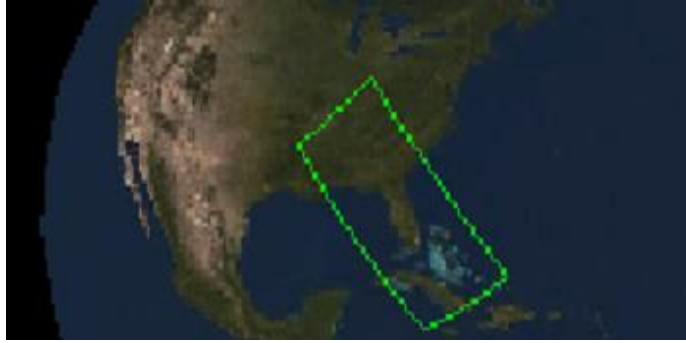


Figure 65: Overview Close up

4.4 ACES Viewer

4.4.1 Basic Infrastructure

4.4.1.1 Visualization Modeling

4.4.1.1.1 Introspection and Analysis

In order to better support several features that have been requested for the Interactive Visualization in 4D plug-in module for the ACES Viewer, we have added several methods that examine the graph structure of a visualization in the ACES Viewer. These methods allow developers to identify relationships between visualization components. Some examples include:

- Identifying Tables that provide data to a particular Renderer or DisplayContainer.
- Identifying all visualization components related to one display, but not shared by others (this allows removal of unused visualization components when closing a display)
- Identifying incoming and outgoing links associated with any single visualization component.
- Generic graph traversal functionality that can be used to walk the entire visualization graph, or a sub-graph. (This can be used for a variety of purposes, and provides the foundation for most of the other functionality listed here).

4.4.1.2 Configuration and Visualization Persistence

We implemented a new persistence system designed to save the state of Java objects to disk for long term storage. This is a replacement for and a significant improvement over the previous TextConverter system. The original system relied on developer-provided TextConverter implementations for each Java Class that was to be persisted. These converters examined the state of the object to persist, and produced a custom textual representation of the object, that was ultimately stored in an XML file. Loading data from the file involved using the same TextConverter implementations to produce a Java Object from the textual data. This had several insurmountable problems:

- It was extremely difficult to represent multiple objects in a single string. When persisting an object that referred to other objects, all of these objects' states needed to be included.
- It was not possible to represent shared objects. If two objects (A and B) to be persisted both referred to a third object (C), there was no way to know that the third object should be shared. When loading this data, four objects would be created - two representing the state of A and B, and two identical objects representing the state of C.
- TextConverter implementations had to support all subclasses of the class they are responsible for converting. Since type information was not directly stored in the XML files, the only way to determine which TextConverter to use to decode a converted object was to use the data type of the referring property. The property types are frequently a super class, or interface type of the actual object type used as the property values.
- TextConverter implementations for complex types were large and error-prone. Developers had to ensure that the conversion worked the same both ways (to and from text). They also had to design a textual representation for each converter that would work for that type, and not interfere with the persistence of other types (when converting nested object hierarchies).

The new persistence system addresses all of these problems, and dramatically simplifies the conversion process implementations, eliminating the need for any explicit per-type conversion support in many cases.

- To simplify the converter implementations, we designed an XML based file format that consists of a sequence of encoded object declarations, constructor calls, method calls, and field access operations. A custom converter emits a sequence of these statements that are used to configure a single object. No custom conversion is necessary to load the data; instead a generic decoder reads the statements and executes them one by one to restore the original object.
- The persistence system automatically traverses the object graph. Whenever a converter emits a statement that refers to a non-primitive object, the converter for that object's type is used to generate its persistent representation. Unlike the previous system, each converter only needs to handle a single object, and not any referenced objects.
- The system keeps track of which objects have already been persisted. If that object is encountered again, a reference to it is stored in the XML file, rather than generating statements to persist another copy of it. This allows shared objects to be correctly represented in the file.
- Converters can support a single type, or a hierarchy of types. When registering a converter to handle a particular Java type, users can specify whether the converter can only handle the specific type(s) that it was registered for, or whether it can also handle sub-classes. Even when a converter is registered to handle sub-classes, another converter can still be registered to override support for specific sub-types.
- Built-in support for common data types. The system provides converters for common Java types including various collection types, fonts, points, etc. It also provides a generic converter for JavaBeans that will automatically generate statements for types that have a no-argument constructor and adhere to the JavaBeans specification.

This system provides significantly more flexibility than the previous system, with a lower development cost.

4.4.1.3 Module and Plug-in Framework

For some time the ACES Viewer has relied on the Java Plug-in Framework (JPF) to facilitate modularity in the various subsystems of the application, and to provide a mechanism by which certain aspects of the ACES Viewer could be extended. As the ACES Viewer continued to grow and started being used as a platform for visualization of data sources other than the ACES output databases and files, we began to separate the various subsystems into finer grained modules. Each module was designed with a single purpose, making it easier to locate code and documentation for specific tasks, and reducing inter-module dependencies. We also made many of the modules optional so that developers building an application based on the ACES Viewer could pick and choose the functionality that their application needed, and disregard unnecessary modules. For example, one module provides access to Center/Tracon Automation System (CTAS) message data, which is only useful to researchers working with that data type, and need not be available in the User Defined Operational Picture (UDOP) application. We also made the ACES Viewer application GUI module(s) optional, so that each application can present their own user interface if desired. This is where we ran into problems with the Java Plug-in Framework.

It is not possible (without modification to JPF) to define an extension of a module that is not present in the application. For example, the JDBC module provides access to relational database data, and declares an extension of the ACES Viewer user interface (workspace.ui) module to provide a wizard interface to specify database connection parameters. If the workspace.ui module is not present, the JDBC module should still be capable of providing its primary functionality, accessing database data, only the wizard user interface components will not function. When JPF initializes a module, it checks all of the extensions, and if the module declaring the extension point is not present, it fails with an error.

It was also nearly impossible to launch the ACES Viewer in an Integrated Development Environment (IDE) using JPF due to the way that it searches for module metadata. This has been a long requested feature by developers working with the application so that they could take advantage of the rapid code/test/debug cycles made possible through IDE tooling. Finally, the module loading and isolation functionality provided by JPF was becoming difficult to use and required explicit management of Java's ClassLoader system throughout the ACES Viewer, and was beginning to suffer performance problems as the number of ACES Viewer modules continued to grow.

These problems as well as other problems led to the replacement of JPF with a more sophisticated module and extension management system, the Open Services Gateway Initiative (OSGi). OSGi is a very detailed API and metadata specification for modular systems, originally designed for embedded applications, but now used for desktop and server applications as well.

The existing ACES Viewer code base required very few changes to switch from JPF to OSGi. Most of the changes were related to differences in module metadata formats. The change provided several immediate benefits:

- Optional modules and extensions have well defined behavior in OSGi, and we were able to quickly develop alternate user interfaces that leverage the core ACES Viewer modules and do not use the ACES Viewer UI modules.

- OSGi uses more sophisticated class-loading routines that provide better performance over JPF, while simultaneously eliminating the need for application code to manage the ClassLoader system.
- The OSGi environment is supported by major Java IDEs (Eclipse in particular, is itself built upon OSGi) which allows ACES Viewer applications to be launched and debugged directly from the IDE. Eclipse also provides graphical interfaces for specifying module and plug-in metadata. The application can still be developed and used without the assistance of an IDE if desired, with no more complexity than before the change.
- Using OSGi allows us to use any of the thousands of existing OSGi modules, including several that are of particular interest to the ACES Viewer, such as graphical editors for Groovy scripting.

The module metadata is split into two parts with OSGi. The first part is the *bundle manifest* which describes the content of a module, any inter-module dependencies, and any private libraries used. This metadata is incorporated into the standard Java jar manifest using key/value pairs specified in the OSGi documentation. The second part is the extension point and extension declaration file. This defines any extension points provided by, or used by a particular module. This is defined in a separate XML file, where the majority of the XML elements are defined by the module that specifies the available extension points.

To simplify implementation of new modules, we now provide a template project containing a default directory structure, build file, skeleton bundle manifest and plugin xml files, and project definitions for use with the Eclipse IDE.

4.4.1.4 Dependency Injection

The ACES Viewer's visualization components have used Java 5 annotations to mark certain methods as input parameters. These annotations were used by the visualization graph component to determine how visualization components can be linked together. During the course of this effort, we formalized this system and developed a more robust and feature rich framework to simplify its usage, and incorporate new functionality. The new framework is based on well understood design principles for dependency injection systems. The ACES Viewer implementation is generic and can be reused for other applications that may benefit from dependency injection.

4.4.2 Rendering and Visualization

4.4.2.1 General Enhancements

4.4.2.1.1 Frustum Culling

Several improvements that we made to JView have simplified the process of determining whether a particular visual element in a 3D scene is visible from the camera. This has allowed us to augment the majority of the ACES Viewer renderers with support for frustum culling. By not submitting visual elements that would not be visible on screen to the graphics hardware, we reduce the system bus bandwidth needed to render a scene and improve performance.

4.4.2.1.2 Level of Detail Controls

To better support lower end computer hardware, we added Level of Detail infrastructure to the ACES Viewer. Users specify the desired Level of Detail for a particular scene as a value on a continuous scale between best performance, and best quality. Each visualization component associated with the scene that supports Level of Detail is provided with the user setting, and automatically changes any rendering settings that affect performance based on the value. This system only provides coarse-grained control over rendering quality, however it is easy to use, and provides a future opportunity to automatically adjust performance-sensitive settings based on the host system's capabilities.

4.4.2.2 Display Containers

4.4.2.2.1 3D Scene

We have added several new capabilities to the JView based 3D Scene container, and exposed these and other capabilities for manipulation via the user interface.

- **Light Follows Camera.** This option specifies whether the direction of the light in a 3D scene is fixed, or tracks the viewing direction.
- **Camera Elevation Restricted.** By default, the camera cannot be moved below the ground plane. This is particularly helpful to new users that are not familiar with the navigation controls, because it is more difficult to move the camera to potentially confusing positions. Disabling this restriction allows advanced users to move under the ground plane to view elements from underneath.
- **Bounding Box Displayed.** Mostly used for debugging, this shows the bounding box of all of the elements present in the scene.
- **Capture Alpha.** Allows the alpha color channel (translucency) to be saved when capturing images or movies of a 3D scene. This option is disabled by default because it requires an alpha visual to be selected for rendering, which is not available on all platforms, and can cause unexpected behavior on others.

- Stereo Mode. Enables anaglyph or side-by-side stereo pairs to be used when rendering the scene for 3D visualization. Disabled by default since this requires special hardware (for active stereo), or anaglyph glasses for anaglyph mode.

4.4.2.2.2 Tabular Scene

We incorporated the 2D tabular visualization from the User Defined Operational Picture project to provide a configurable view of the raw data from any data source or transform in a visualization graph. This is shown in Figure 66.











Name	Data	Center Latitude	Center Longitude
ALABAMA		32° 47' 43"	-86° 49' 40"
ALASKA		64° 37' 38"	-152° 55' 12"
ARIZONA		34° 17' 35"	-111° 39' 48"
ARKANSAS		34° 54' 31"	-92° 26' 19"
CALIFORNIA		37° 15' 27"	-119° 36' 41"
COLORADO		39° 00' 03"	-105° 33' 05"
CONNECTICUT		41° 37' 27"	-72° 43' 34"
DELAWARE		38° 59' 27"	-75° 30' 18"
FLORIDA		28° 39' 10"	-82° 28' 33"
GEORGIA		32° 39' 12"	-83° 27' 06"

Figure 66: Tabular Scene displaying data from the US Map data source.

4.4.2.2.3 JFreeChart

Charting is a useful tool for any type of analysis. Even with advanced visualization sometimes a graph is the best way to convey information. Recently we added JFreeCharts to the ACES Viewer which gives the ACES Viewer the ability to add many types of graphs. The ACES Viewer can now create line graphs, bar graphs, pie graphs, block graphs, and candlestick graphs. Block graphs and candlestick graphs are uncommon. Block graphs are graphs that have three inputs: an X, Y and intensity. Block graphs draw each (x, y) as a block and then colors that block based on the intensity. Candle stick graphs have five values: time period (X), open value, high value, low value, and close value. The candle stick will draw a line on every time period from high to low and then a bar from open to close which represents a candle stick. In the diagrams below there are five diagrams showing the different charts (Figure 67, Figure 68, Figure 69, Figure 70, and Figure 71).

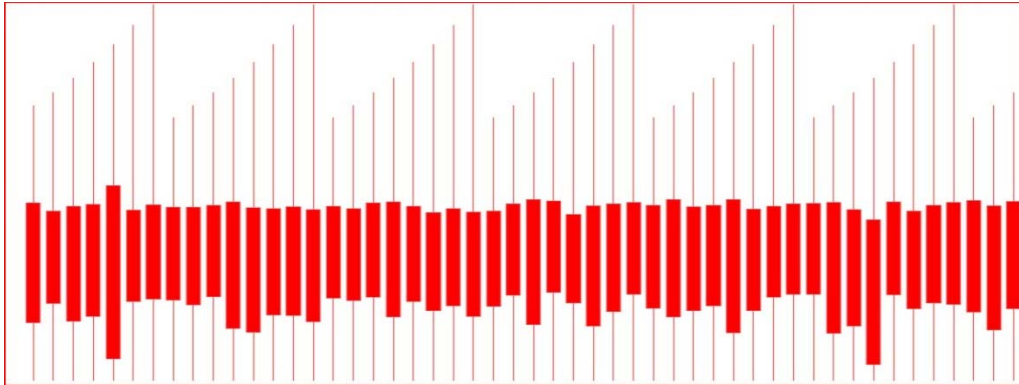


Figure 67: Candle stick renderer

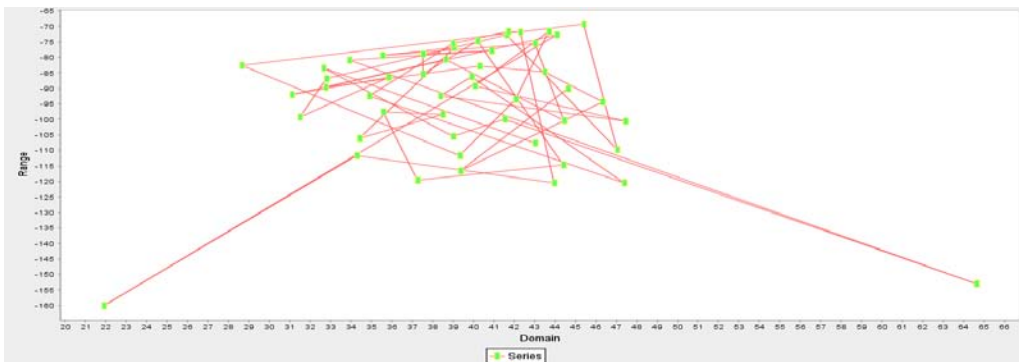


Figure 68: XY Line and Shape graph

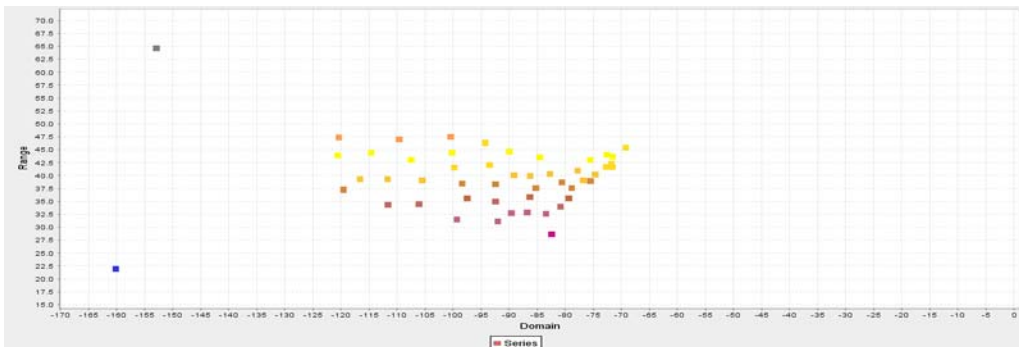


Figure 69: XY Block Graph

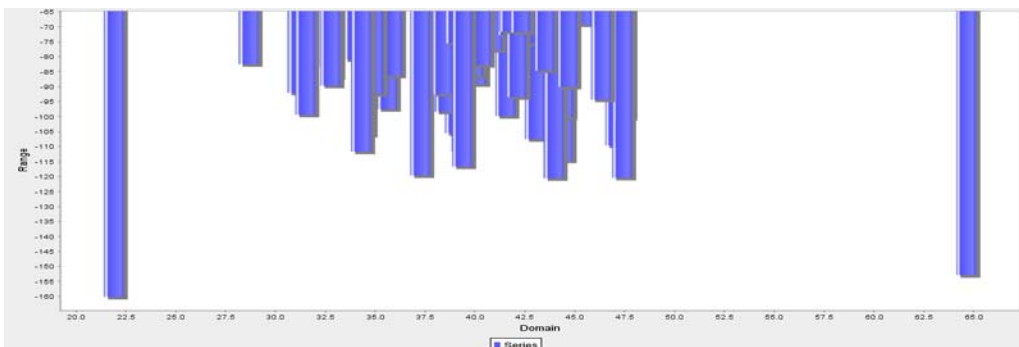


Figure 70: XY Bar Graph

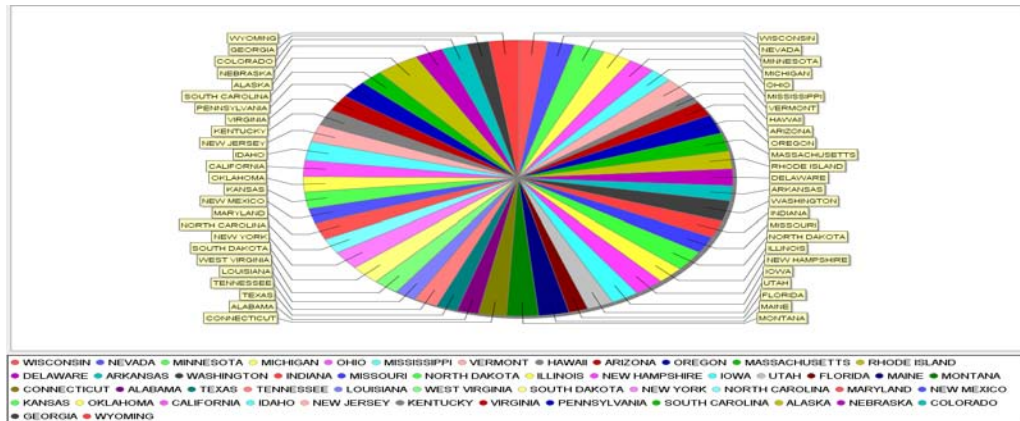


Figure 71: Pie Graph

4.4.2.3 Renderers

4.4.2.3.1 Polygon Renderer

We incorporated support for lighting into the ACES Viewer's Polygon Renderer. Lighting provides a simple means by which users can easily discern changes in orientation of the faces of each polygon, making it easier to see the shapes. We have also added projection accurate per-vertex normals to the top face of the polygons, so the lighting effect is rendered correctly for non-planar projections.

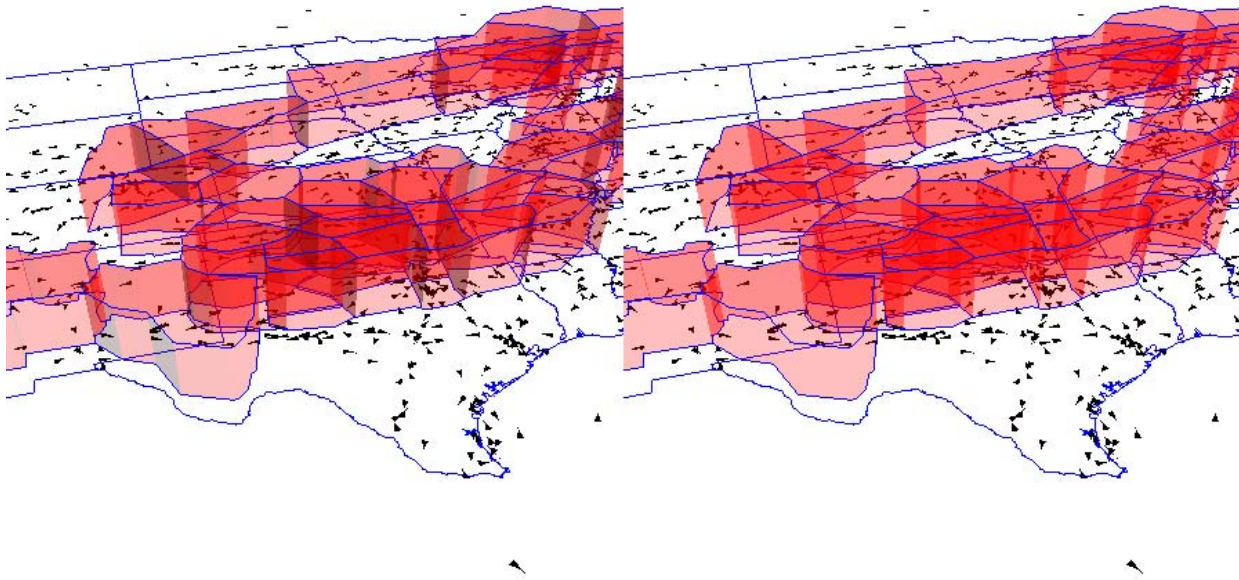


Figure 72: The Polygon Renderer with lighting (left) and without lighting (right)

4.4.2.3.2 Histogram Grid Renderer

We addressed a problem with the Histogram Grid Renderer that was occasionally causing VM crashes on some systems. The problem was related to the fact that the renderer frequently allocated new direct memory buffers used to transmit vertex data to the graphics hardware. Previously allocated buffers were discarded, and it was assumed that Java's built in garbage collection mechanism would reclaim the memory used by the discarded buffers. Unfortunately, due to the slightly different way that directly allocated memory is managed by the VM, rapid allocation and deallocation of buffers caused the VM to throw an `OutOfMemoryError` (and subsequently crash) even though there was still enough memory eligible for reclamation by the garbage collector. We believe that the garbage collector was only taking the amount of memory used for the reference to the memory (a few bytes) when determining whether there was enough memory available for new allocation, rather than taking the entire buffer size (typically hundreds to thousands of kilobytes per buffer) into account. We fixed the problem by significantly decreasing the memory allocation frequency of the Histogram Grid Renderer, and reusing existing buffers where possible.

We later encountered this problem with several other JView based components, both in the ACES Viewer and in JView itself. We developed some utilities to better manage buffer allocation in JView to address this problem that (among other things) will request a full garbage collection pass if allocation fails. This approach fixes the problem for most situations, but in general we found that it is best to minimize frequent allocation of buffers, and instead reuse existing instances where possible.

4.4.2.3.3 Volume Renderer

We adapted the existing slice-plane based volume renderer to achieve pre-integrated volume data classification with lighting. The new implementation takes advantage of newer capabilities of the OpenGL API (version 2.0) to perform volume classification directly on the graphics hardware. This approach differs from the previous in that instead of sending color and opacity down to the graphics hardware for rendering, we send the numeric volume data. This permits interpolation of the values in the data domain instead of the color domain, leading to improved rendering quality without the 'blocking' artifacts seen with the previous pre-classification approach. This also allows us to calculate normals for each rendered volume sample (using central differences of the interpolated data values) for lighting effects. Figure 73 shows the difference between the previous implementation (left) and the new implementation (right) for a sample volume. Note the improved 'smoothness' of the picture on the right, and the more accurate color representations.

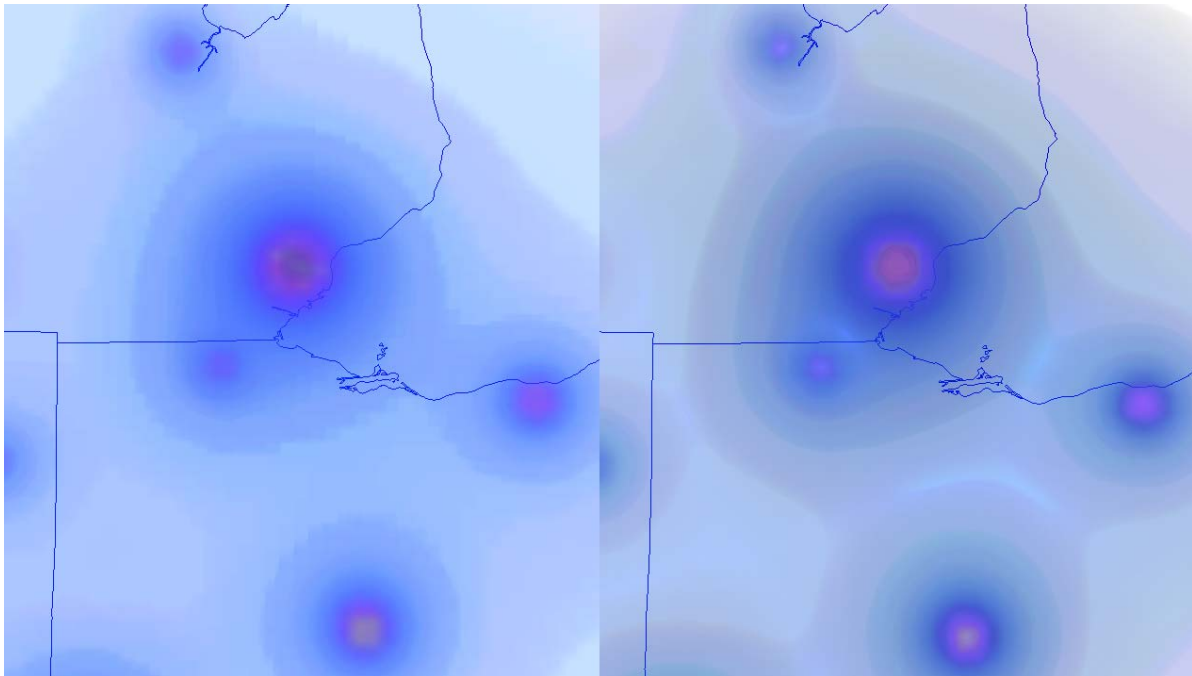


Figure 73: Comparison of old pre-classification renderer (left), and new pre-integrated renderer (right)

Figure 74 shows a comparison of the old and new volume renderer implementations where a single value in the data is mapped to an opaque value, producing an iso-surface representation of the data. The lighting capabilities of the new implementation provide a clearer representation of the data.

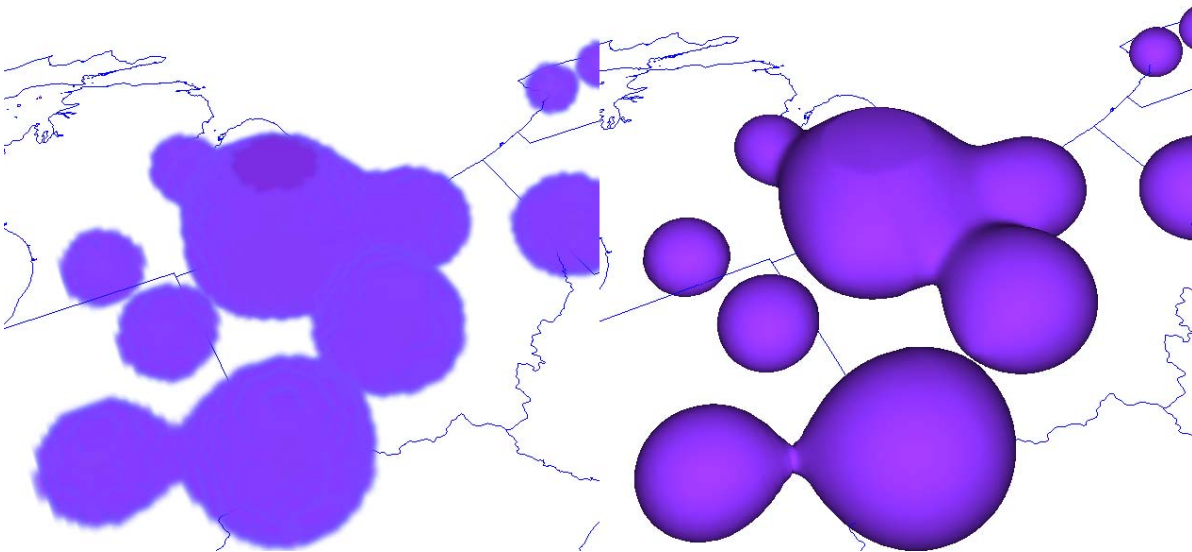


Figure 74: Comparison of iso-surfaces rendered using the old (left) and new (right) volume renderers

4.4.2.3.4 Model Renderer

We dramatically improved the performance of the ACES Viewer's Model Renderer using newly available OpenGL shader program techniques. Geometry Instancing allows a model's geometry to be transferred to the graphics card once, and rendered many times in different positions and with different visual attributes. This takes advantage of relatively recent OpenGL functionality, so we check to ensure that the host system supports this method of rendering, and fall back to the previous method if it is not. This optimization has allowed us to render more than 300,000 paper planes in a scene while maintaining interactive frame rates on our development system, compared to the previous method which could only render 15,000-20,000 at the same frame rate. While we may not actually need to render that many objects in a single scene, the optimization reduces the load on the graphics hardware, the CPU, and the PCI bus, leaving more resources available for other application and rendering functions. The basic functionality developed to enable this level of performance has been incorporated into JView.

We also improved the level of detail support for the Model Renderer, allowing lower quality proxy geometry to be used in situations where the rasterized representation of the geometry would be nearly identical if the full quality geometry was used. The improvements include better determination of the quality level that a particular geometry should be rendered at (based on the screen area that it will occupy), and API design improvements that make it easier to implement geometry that supports level of detail.

Taking advantage of the new picking capabilities in JView, we added support for users to select a model in a 3D scene and automatically follow it with the camera as it moves around. This function was specifically requested by NASA researchers to facilitate movie generation and capture.

Finally, we added the ability to anisotropically scale the models based on the global altitude scale that can be set for the model renderer. This is important to ensure that geometry that is intended to represent a physical volume actually represents the correct volume in the scene. For example, researchers frequently visualize the minimum permitted separation distance between aircraft as cylinders surrounding the aircraft model. If this is done with altitude scaling set to a value other than 1, the visualization will show exaggerated vertical distances between the cylinders. If the cylinders are not scaled accordingly in the vertical direction, then they may appear to be completely separate, when in fact they should intersect. Figure 75 shows examples of this vertical scaling.

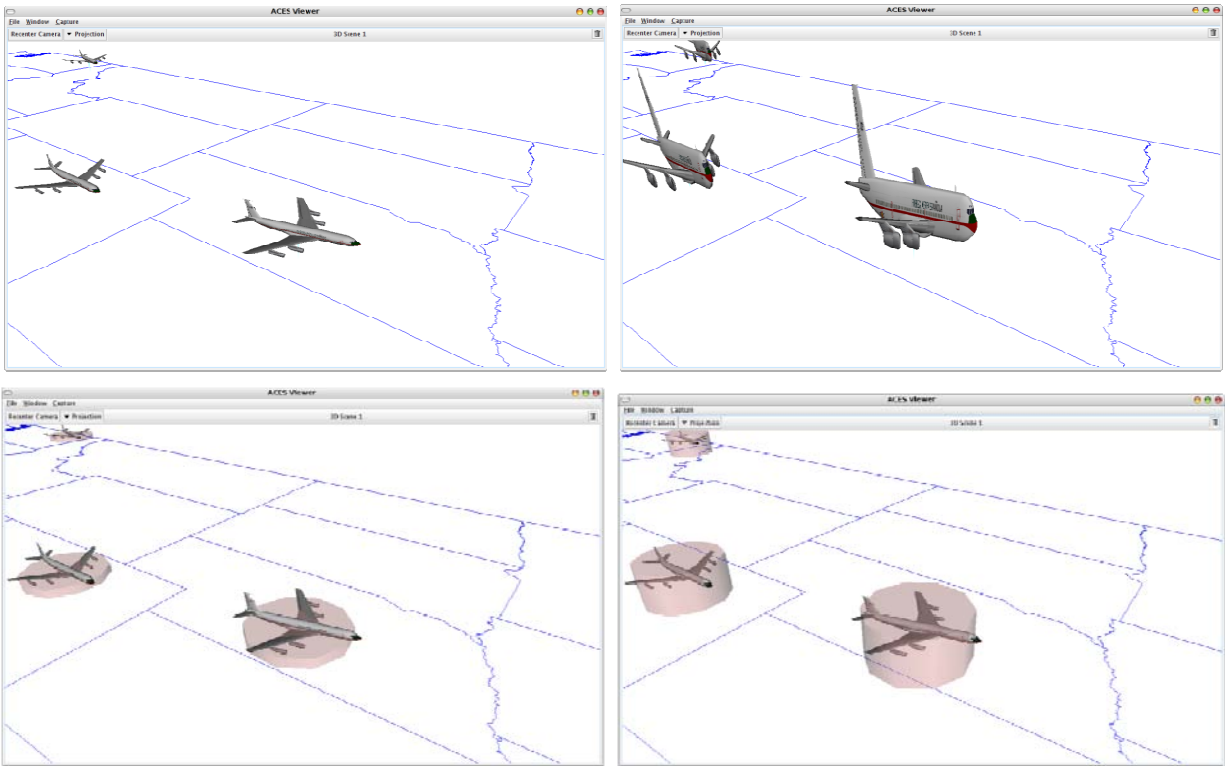


Figure 75: Anisotropic Scaling of Model Geometry in the Model Renderer

4.4.2.3.5 Line Renderer

We made two significant improvements to the LineRenderer. It now supports per-vertex coloring of lines, allowing a single line to have multiple colors across its length (Figure 76), where previously a color could only be applied to an entire line. We also added support for direct rendering of point-location data, where previously, this type of data first needed to be grouped and sorted into explicit collections of points for each line using an external Transform component.

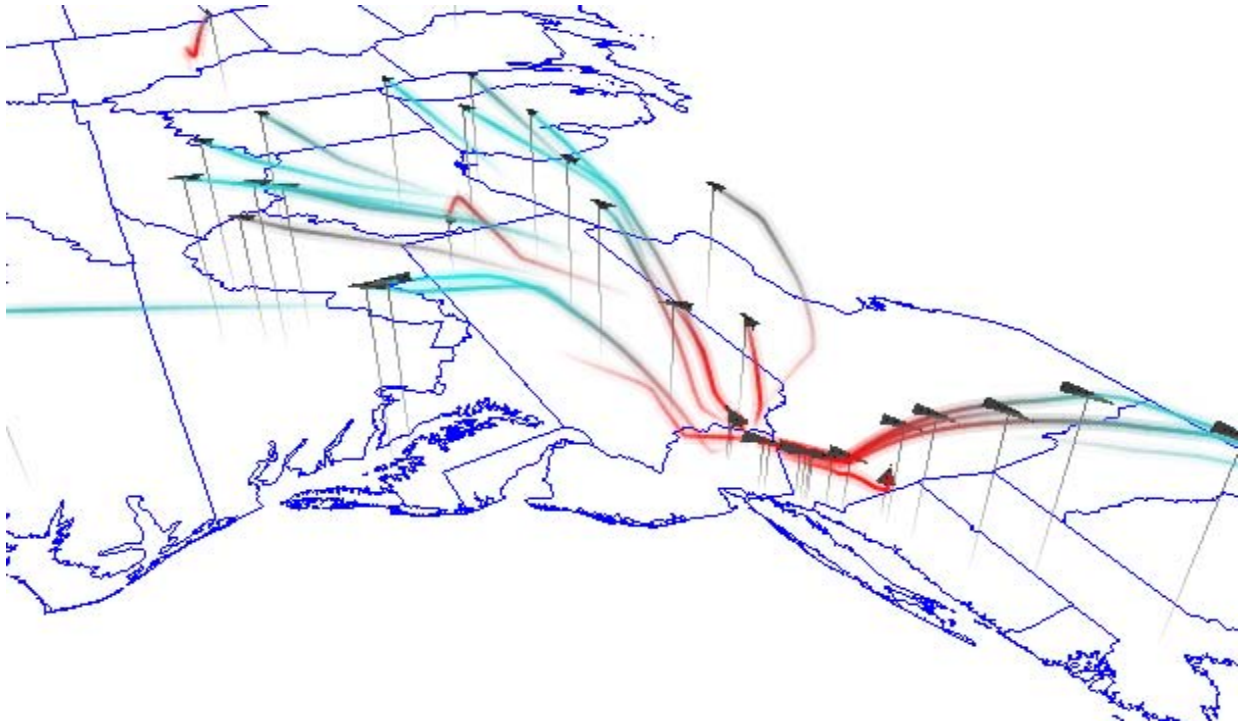


Figure 76: The Line Renderer showing aircraft trajectories colored based on altitude

We also improved the performance of the Line Renderer by 2x-3x when using an alpha gradient at the beginning and/or end of the line segments. This was achieved by using OpenGL's 1D texture mapping capabilities to apply the alpha gradient, rather than per-vertex colors sent to the graphics hardware for each render.

4.4.3 Data Access

4.4.3.1 Memory Consumption and Efficiency Enhancements

As the Airspace Concept Evaluation System (ACES) simulation development advances, researchers are running more complex simulations and generating larger amounts of data. To support visualization of this growing data, we reduced the overhead and improved the memory efficiency of several components in the ACES Viewer.

For several components that deal primarily with numeric data (particularly the Line Renderer and Time Filter Transform), we reduced overhead by restructuring the way data was stored. Among other improvements we now use primitive arrays for most numeric data rather than Object arrays which incur additional reference overhead (that is typically larger than the data being stored). We also designed a new TupleStore API to permit pluggable data structures for storing Tuple data. We provided several implementations of this API with various performance and memory consumption characteristics that can be used by Table and Transform developers.

For situations where optimization of the data structures is not enough, we now provide a Binary Table implementation and API that simplifies the process of saving Tuple data to disk for out-of-core access. Like the CSV Table, this Table implementation loads data from the file on demand and supports random access to the Tuples it contains. It supports a pluggable serialization mechanism that permits developers to specify how data should be written to disk. It provides the advantages of faster data serialization/deserialization and smaller file sizes compared to CSV data, due largely to the binary data format. The Binary Table implementation has been incorporated into several existing Table implementations (including JDBC Table and JDBC Query Table) to provide the user with the option of caching the data to a local file (which not only reduces memory consumption, but can provide faster data access in some situations).

4.4.3.2 CSV Data

To support access to and visualization of arbitrary data stored in CSV files, we implemented a generic CSV Dataset. The associated wizard allows users to select any number of CSV files, and specify a table name and schema for each. Based on the user provided schemata, the dataset parses the associated CSV file, converting the textual content to the appropriate Java data types.

4.4.3.2.1 Usability

We implemented functionality to attempt to automatically derive a Schema from a CSV file. The Schema is comprised of three qualifiers for each field in the data - a name, a data type, and an optional data unit specification. When the CSV file contains a machine readable Schema declaration (as all CSV data output from the ACES Viewer does), then all of these qualifiers can be determined for all fields. If the CSV file does not contain the Schema declaration, the names and types for each field may still be determined. Field names can be defined in a CSV file by placing the names separated by commas in a comment line in the file. Since the CSV specification does not define conventions for naming the columns, we selected the comma-separated format based on conventions used by ACES developers. Field types can be inferred by attempting to parse the data from each field in each row. All of the available parsers are given each field to convert to the parser-specific data type. If parsing fails, the failed parser is removed from the list of candidates for the column. When the entire file has been processed in this manner, a relatively small list of potential data types for each field (ideally a single type, but this is not always the case) is presented to the user for further refinement. Any fields that are not parsable by any of the available parsers are treated as textual strings.

The CSV Dataset uses this new functionality to assist users when importing CSV files. In the best case, users do not need to specify any information except for the location of the CSV file, and in the worst case, they may have to specify column names, and select data types from a short list for each column.

4.4.3.2.2 Efficiency

In order to reduce the application memory utilization when using large CSV file data sources, we implemented a new file reader designed to load data directly from disk, as it is needed. This works as follows:

- When a CSV file is opened, the file is scanned once to identify the byte offsets of each record in the file. The offsets are recorded in memory for every *n*th record, where *n* is a configurable value.
- When reading a CSV record from the file, a small section of the file (near the record being read) is mapped into application memory. The character data is decoded and the resulting string is split into fields.
- Each field from a record is converted into a Java object when it is accessed. This avoids potentially time consuming conversion of fields that are never read.
- When reading another record, it is possible that the corresponding section of the file is already in memory (particularly when reading sequential records). In this case the memory mapping operation can be skipped. Otherwise the required file section is loaded and the process repeats.
- Records are also lazily loaded when iterating over the entire file. No data is actually read from disk until a field is read from a record. Like lazy field conversion, lazy record loading avoids reading data from disk that will never be used.

The new CSV reader provides sufficient performance to load records directly from a disk, pass the data through a visualization network and render the data with interactive frame rates, while simultaneously reducing memory utilization to a trivial level.

4.4.3.2.3 CSV to RDBMS Experiment

In order to improve (decrease) memory utilization when accessing data from large CSV files, we investigated the possibility of converting the data in a CSV file into a relational database, which could then be accessed efficiently using the same JDBC mechanisms presently used by the ACES Viewer to fetch data from MySQL. This would provide the benefit of using a low-overhead database management system (Apache Derby in this case) to handle out-of-core data access and query evaluation. Unfortunately, the experiment revealed that there would be a significant challenge in initializing the database with CSV data due to the fact that the mapping from the Java type system to the SQL type system is not straightforward. We also investigated several open source JDBC drivers that provide direct access to data in CSV files; however these were typically less efficient than the existing ACES Viewer systems for reading CSV data.

4.4.3.3 Database (RDBMS) Data

In order to support access to data in relational databases that support ACES, but were not generated by ACES (e.g. simulation input data), we implemented a new dataset extension. The Basic Database Dataset is largely derived from the ACES Database Dataset, but expectations that certain tables were present in the database were eliminated. The ACES Database Dataset was

then refactored to share code as much as possible with the Basic Database Dataset implementation.

4.4.3.4 CTAS Data

We developed a proof of concept ACES Viewer plug-in that provides access to data produced by the Center Tracon Automation System, an operational tool used to assist air traffic controllers. The plug-in presently reads recorded data, allowing it to be visualized within the ACES Viewer. It can be developed further (to view live output from CTAS) if it is found to be useful.

4.4.3.5 Weather Polygons

We added a parser for weather polygon data used by ACES. Weather polygons denote regions of airspace classified by the severity of weather (including wind, precipitation and lightning) within. This data has been provided to us in CSV format, and consists of a sequence of time-stamped lon/lat/alt polygons representing regions with severe weather. Figure 77 shows an example visualization using this data.

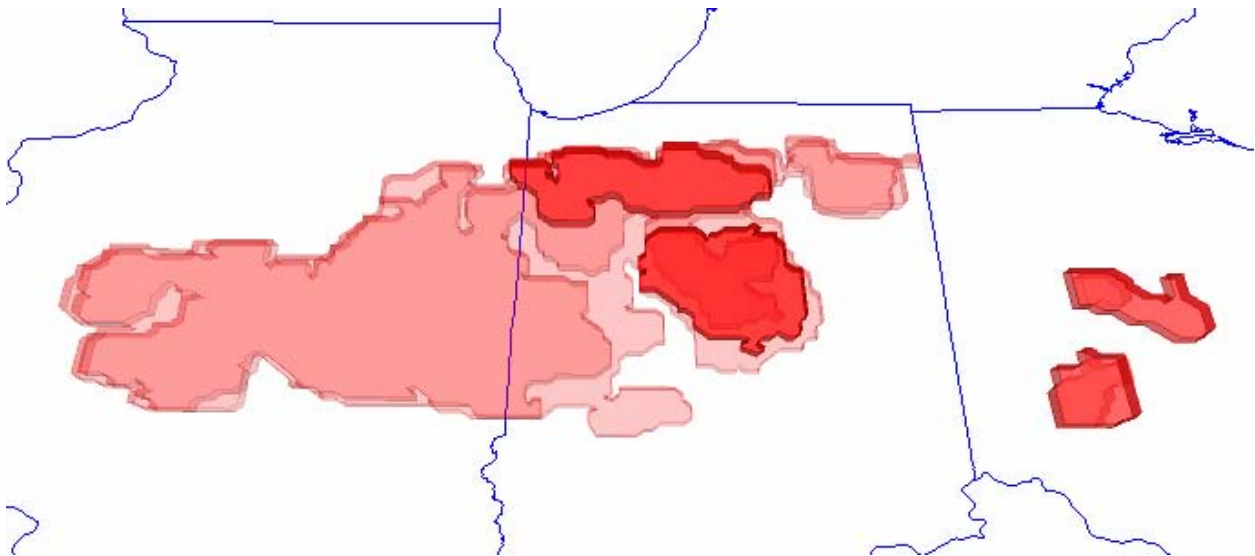


Figure 77: A Visualization of Regions with Severe Weather Denoted by Weather Polygons

4.4.3.6 XML Air Zones

ACES version 6.2 introduced a new format for the airzone boundary definitions. The new format uses XML to specify all sector, sub-sector, and center boundary information. We implemented a parser for the new format, and integrated it into the ACES Viewer's ACES data module.

4.4.4 Data Processing

4.4.4.1 Units

Much of the ACES produced data (altitude in particular) is measured in feet, while most JView elements expect input data to be measured in meters. This means that for a visualization of a simulation run to be accurate, all data must be converted to meters. To this end, we have incorporated generic support for unit conversion into the ACES Viewer. This capability is based on the Java Specification Request #275 reference implementation library, which provides enumeration of a variety of units, as well as capabilities for converting between compatible units.

To simplify the process for developers, unit conversion takes place automatically. Developers incorporating new data types into the ACES Viewer can declare the units of data fields via the ACES Viewer Schema class that describes each data Table.

TupleExpressions are responsible for converting units if necessary. Data consumers (Renderers and Transforms) can optionally specify a target unit for conversion for an expression. If units are specified for both the source data, and the desired unit for the data consumer, values are converted between the units as the data is read from the source.

4.4.4.2 Concurrency

In an effort to take advantage of the fact that most commodity computer hardware now contains multiple general purpose processing units, we have begun to enhance the granularity of parallel and concurrent processing in the ACES Viewer. Previously, opportunities for concurrency were exploited at the visualization component level. Each data access, transform, and renderer component typically performed their own tasks in a single threaded manner, but each component could (where possible) be performing its work concurrently with the other components in the visualization. The effectiveness of this model at taking advantage of the available processing units varied based on the structure of the visualization. Consider the diagram in Figure 78.

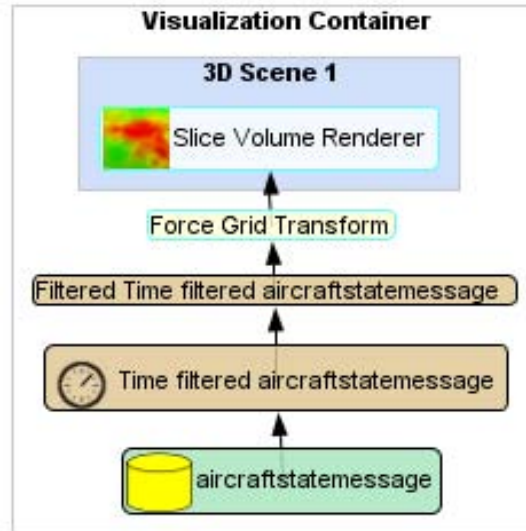


Figure 78: A visualization with linear data flow

Figure 78 shows a visualization comprised of three transforms (Force Grid, Filter, and Time Filter), a Renderer (Slice Volume Renderer), and a data access component (JDBCTable). In this case, there is a linear flow of data from the aircraftstatemessage table to the renderer. This means that the Filter Transform cannot process data until the Time Filter does, and similarly, the Force Grid Transform cannot perform its processing until the Filter Transform does. These data dependencies effectively force the entire data flow to be processed in a single thread. This visualization represents the worst case for concurrency in the ACES Viewer. Figure 79 probably represents a more typical situation.

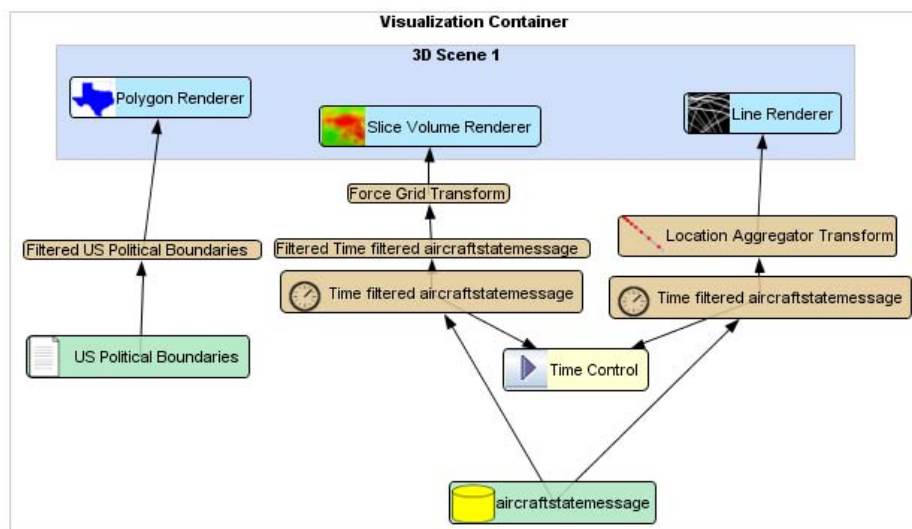


Figure 79: A visualization with parallel data flows

In Figure 79, there are multiple pathways through the processing network to the renderers. Each pathway can be processed concurrently since there are no data dependencies between separate pathways. In this example we can use a maximum of three threads for concurrent processing (equivalent to the number of paths from the data access components to the renderers). After analyzing visualizations produced by researchers and the IV4D extension to the ACES Viewer, we determined that in the average case, the application can use anywhere from 1 to 3 threads for concurrent processing of visualization data.

By default, Tables now issue change events asynchronously using a pool of threads, rather than issuing the events synchronously and sequentially. This removes the need for individual Table/Transform implementations to handle concurrent processing themselves in most situations, while still taking advantage of multiple processors in the host system if present.

To take better advantage of multi-processor systems, we have begun exploring opportunities for concurrency *within* each of the visualization components. The ForceEffectGrid Transform was one of the first that we looked at because it is highly CPU intensive, and the algorithm it uses is easily parallelized. We were able to significantly increase runtime performance of the ForceEffectGrid on multi-processor systems. Performance scales nearly linearly with the number of available CPUs, and tests on a 16 processor machine produced pleasing results. We intend to continue this work with other Transform components in the future, including incorporating Doug Lea's Fork-Join concurrency framework slated for inclusion in future versions of Java.

4.4.4.3 Expressions

4.4.4.3.1 Schema Adapter

Most visualization components that accept input from a Table instance in the visualization graph use expressions to extract relevant data from each Tuple in the Table. There is a significant amount of 'boiler-plate' code required to manage each expression throughout the lifecycle of the Table-consuming component:

- Validating the expression against the input Table's Schema whenever the input Table changes
- Monitoring the expression for changes that might affect the value it returns.
- Providing default values for expressions that are un-specified, or invalid.
- Providing accessor methods for users to change the expressions.
- Ensuring that expressions, when set by a user, conform to the input requirements of the Table consumer (data type and units).
- Using the expressions to 'decode' data in input Tuples.

We introduced a new abstraction called a Schema Adapter to handle all of these tasks automatically. A Schema Adapter is created by the consumer, and initialized with the Schema (data names, types and units) expected by the consumer (this is the output schema of the Schema

Adapter). The consumer can optionally specify default values for each data field in the Schema. When an input Table is provided to the consumer, the Table's Schema is provided to the Schema Adapter as the 'input schema'. The Schema Adapter automatically validates all expressions against the new input schema. User Interface support for the Schema Adapter allows users to specify the expressions used to extract each output field from the input Tuples, without the need for accessor methods for each expression on the Table consumer class. When reading input Tuples, the Schema Adapter provides methods to convert Tuples corresponding to the input Schema to Tuples adhering to the output schema, alleviating the need for the consumer class to handle conversion itself.

Use of the Schema Adapter has significantly simplified the implementation of visualization components that consume Table data, and reduced the chance of errors in duplicated code. In the future, it may be possible to further reduce the complexity of Table consumer implementations by automatically creating Schema Adapter instances when Tables are linked to consumers. From the consumer's point of view, input data would have a fixed Schema corresponding exactly to the information required by the consumer.

4.4.4.3.2 Groovy Script Expression

We improved the integration of Groovy scripts by changing the way that the Groovy programs are invoked by the ACES Viewer. Previously, invocation involved the use of reflection to identify the byte codes to invoke in Groovy programs. We now automatically generate Groovy code that enhances the user-entered scripts, causing them to implement a specific Java Interface. We can then invoke the compiled script using a normal Java call, rather than using reflection. Since Groovy scripts are invoked extremely frequently when they are present in an ACES Viewer visualization (often as much as once per data record, per rendered frame), this has yielded significant performance improvements.

4.4.4.3.3 Python Script Expression

We added the Jython scripting option to the ACESViewer. Jython is a java engine for the Python scripting language. It works identically to the Groovy scripting except that the scripting language used is Python. A sample can be seen in Figure 80 and Figure 81.

```
Jython Script Editor

from java.lang import *
from mil.afrl.rrs.ifsb.jview.graph import Color4f

def evaluate(tuple):
    #
    # Your code here...
    #
    # To access data, use tuple.get(<field-name>)
    # where <field-name> is one of:
    #
    #   simulationTime (type: Long)
    #   latitude (type: Double)
    #   longitude (type: Double)
    #   altitude (type: Double unit: ft)
    #   heading (type: Double)
    #   flightId (type: Integer)
    #   groundspeed (type: Double)
    #   sectorName (type: String)
    #
    # Return a 'Color4f'
    #

    return Color4f( tuple.get(latitude),
                    tuple.get(longitude),
                    tuple.get(altitude),
                    tuple.get(heading))
```

Figure 80: Jython Code Sample

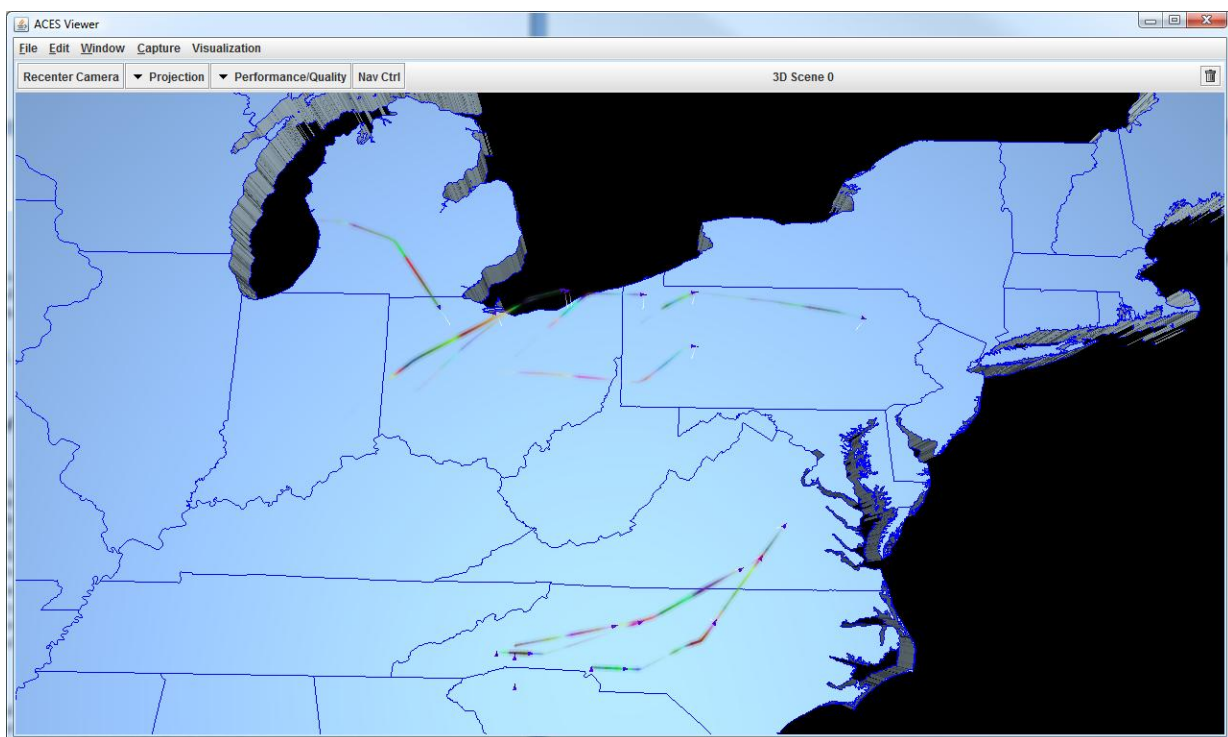


Figure 81: Output from Jython Code Altering Line Color

4.4.4.3.4 Text Expression

We implemented a new Expression type with an associated user interface for functions that produce textual data. The new Expression allows users to specify a combination of literal string data along with textual representations of any number of columns from the input Tuples. This is significantly more flexible than the previous implementation of a text expression which only allowed a single input column to be used, and no literal information. The new Expression uses a specially formatted string where tokens like `${column-name}` are replaced with a textual representation of data taken from the specified column-name in the input Tuples. To simplify use of these tokens, the user interface for editing the new text expression contains a menu that presents a list of the available columns, and inserts the appropriate token when a menu item is selected (Figure 82).



Figure 82: The user interface for specifying a Text Expression

4.4.4.4 Transforms

4.4.4.4.1 Force Effect Transform

The Force Effect Transform computes a discretely sampled 3D field based on the inverse square of the distances from a set of locations to each sample point. This Transform is primarily used to provide a qualitative view of spatial congestion when applied to aircraft locations.

We parallelized this to take advantage of the fact that most modern computer hardware contains multiple CPU cores. The most difficult challenge in this process was not designing the algorithm for parallel computation of the force grid, but rather making the whole process asynchronous so that it did not block other threads that could potentially be used for other processing tasks. After making these changes, the ForceEffectTransform was no longer the performance bottleneck in the visualizations using it. We evaluated the scalability of the new algorithms on machines with as many as 16 processor cores, and they were able to make effective and efficient use of all of the available processors.

4.4.4.4.2 Grouping Transform

We implemented a new generic Transform component that mimics the functionality of a SQL *group by* clause. The Grouping Transform uses two user provided parameters (Figure 83), a grouping criterion and aggregate function. The Transform operates by collecting all input Tuples into sets based on the grouping criterion, and for each set, evaluates the aggregate function over

all of the members to achieve a single aggregate value for each set. Currently, implemented aggregate functions include count (counts the number of members in each set), average, sum, min, max (which operate on numeric values derived from set members), and array (which collects all set members into an array of values).

Figure 83: An early user interface used for manipulating the Grouping Transform

4.4.4.3 Projection Transform

The projection transform provides support for a relational projection operation on input Tuples to produce an equal number of output Tuples with a modified Schema. Possible operations on the input data include removing columns, renaming columns, and deriving columns using the available expressions in the ACES Viewer (Figure 84). For example, a user might wish to associate a color with input Tuples that contain location and velocity information. Using the Projection Transform, the user might write a Groovy script to derive a Color value based on the velocity information from the input Tuples.

Figure 84: The user interface used for editing parameters of the Projection Transform

4.4.5 User Interface

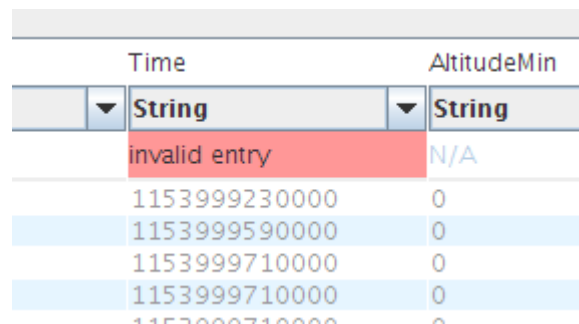
4.4.5.1 Basic User Interface Components

4.4.5.1.1 Overwrite Confirmation

There are several types of data that can be saved to disk from the ACES Viewer, including visualizations, dataset definitions, database connection parameters, and CSV representations of Tables. We added a subclass of Java's built-in JFileChooser user interface component that automatically requests confirmation before overwriting an existing file with the same name as the data being saved. This component is now used in all cases where a user can save to disk.

4.4.5.1.2 Effects Library

We created a small library of generic user effects to help provide feedback to users under certain conditions. This library provides convenient methods to quickly apply common effects to user interface components. One example of these effects is the error indicator, an effect that makes a component smoothly transition from its original color to red when an error is detected in user input for the component. This is shown in Figure 85.



Time	AltitudeMin
▼ String	▼ String
invalid entry	N/A
1153999230000	0
1153999590000	0
1153999710000	0
1153999710000	0
1153999710000	0

Figure 85: Invalid Entry Indicator

4.4.5.1.3 Busy Indicator

Due largely to the fact that the ACES Viewer accesses data from many different sources, including network resources, databases, and local files, there are many times where the application is doing work in the background, but is not updating the user interface. To provide some feedback to the user to indicate that the application is working, we implemented a Cursor Manager that will replace the typical mouse pointer with a platform specific busy cursor, and restore the original when work is complete.

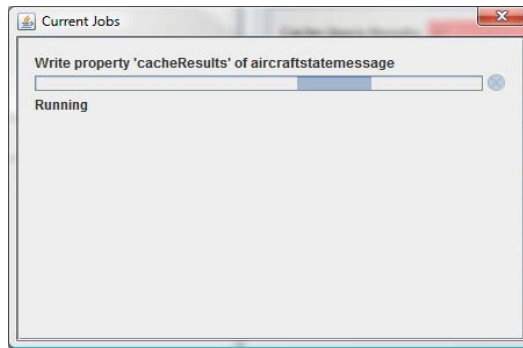


Figure 86: Current Tasks window

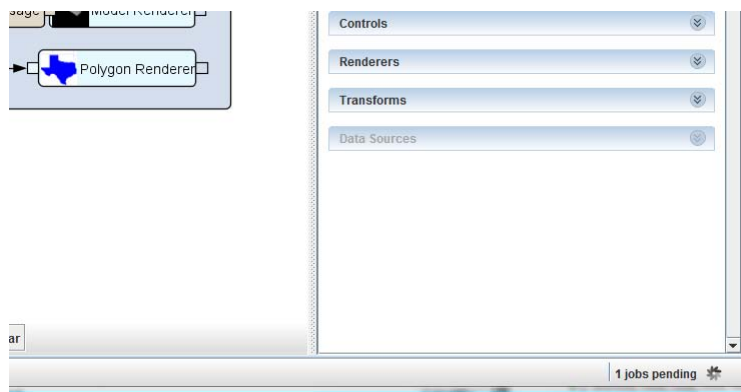


Figure 87: The Task Status Icon

We also developed a persistent status indicator that appears near the bottom of the user interface (Figure 87). When tasks are executing in the background, the status indicator animates, and can be clicked to display a list of tasks that are presently running in the background. The task list shows the progress of each task, and provides a button that allows users to cancel it.

4.4.5.2 Docking Frames

The User Defined Operational Picture (UDOP) effort included development of an improved user interface for the application, based on dockable panels for each major component. This provided a single window UI for the application by default that could be rearranged to suit user needs. We incorporated this work into the core ACES Viewer user interface and resolved some usability issues that were present in the original version. We also added capabilities into the ACES Viewer that facilitate subtle differences in user interface behavior to support the differences between the two applications. These changes will allow future user interface improvements to the UDOP project to either be isolated to that application, or available to all ACES Viewer based applications at the developers discretion in the future, to reduce

fragmentation and promote code reuse. An example of the new user interface is shown in Figure 88.

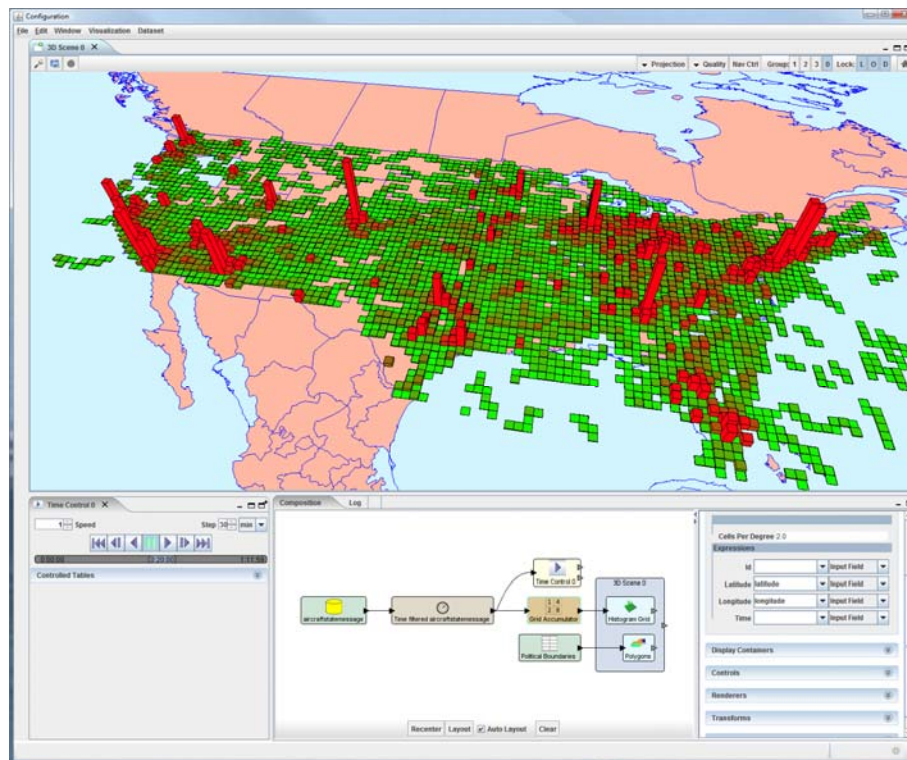


Figure 88: Docking Frames based User Interface in the ACES Viewer

4.4.5.3 Visualization Controls

4.4.5.3.1 Time Control

In response to user requests, we have added several capabilities to the ACES Viewer's Time Control:

- A units combo box so that users can specify the step amount in milliseconds, seconds, minutes or hours.
- A list display to show which Tables are being controlled by the TimeControl. The list also displays details of the tables, including the current simulation time and extent.
- A stop button next to each entry in the controlled Tables list to temporarily disable a controlled table.
- The pause button next to each controlled table allows users to lock the table to the current playback time, and to stop that table from advancing with playback.

The new Time Control is shown in Figure 89.

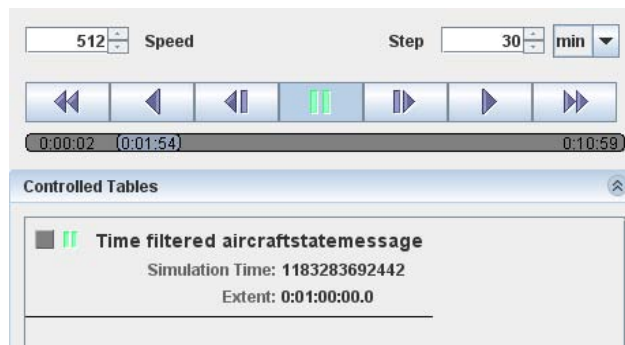


Figure 89: Time Control with buttons to disable or pause each controlled table

4.4.5.3.2 Navigation Control

The 2D navigation control (Figure 90) presents a Java Swing based user interface to control the camera viewpoint in a 3D scene. This component consists of two sections; on the left, there is a map view that shows the camera look-at point (marked by a green 'x'), and the view footprint. Users can click a location on the map to move the look-at point, or can click/drag to specify a region of interest. After specifying the region of interest, the camera is moved so that the entire region is visible in the scene. On the right side of the navigation control there are 3 sliders. These control the spherical azimuth, elevation, and distance of the camera relative to the look-at point. All of the components automatically update when the camera is moved (either with the navigation control, or through direct interaction with the scene) to indicate the actual camera location. This component is designed to give more precise control over the camera position than can be achieved with direct interaction with the scene, particularly when capturing movies, or running the ACES Viewer on lower powered hardware.



Figure 90: The 2D Navigation Control

4.4.5.4 Datasets

4.4.5.4.1 Dataset Definition

We made several improvements to the Wizard framework that provides user interface components to allow users to define Datasets. First, the Wizard framework library previously used by the ACES Viewer is no longer actively maintained. We migrated the existing code in the ACES Viewer to use an alternate framework that had been developed for the NetBeans Integrated Development Environment, and is still being actively developed. The new framework is similar enough to the previous library that only minor changes were required in the ACES Viewer code base.

We also changed the flow of the Wizard graphical user interface components that are presented to the user. Previously, users would activate the wizard, and enable/disable each of the available dataset extensions. The wizard would then proceed to take the user through the steps of defining dataset parameters for each of the selected datasets. This process had some drawbacks. Users were only able to specify one set of parameters per dataset extension. If, for example, a user wanted to create a dataset with two sets of boundary files and a single ACES database, this was not possible. It was also a little unintuitive that users could create datasets with seemingly unrelated data, for example, today's wind speeds with an ACES database that was generated on a different day.

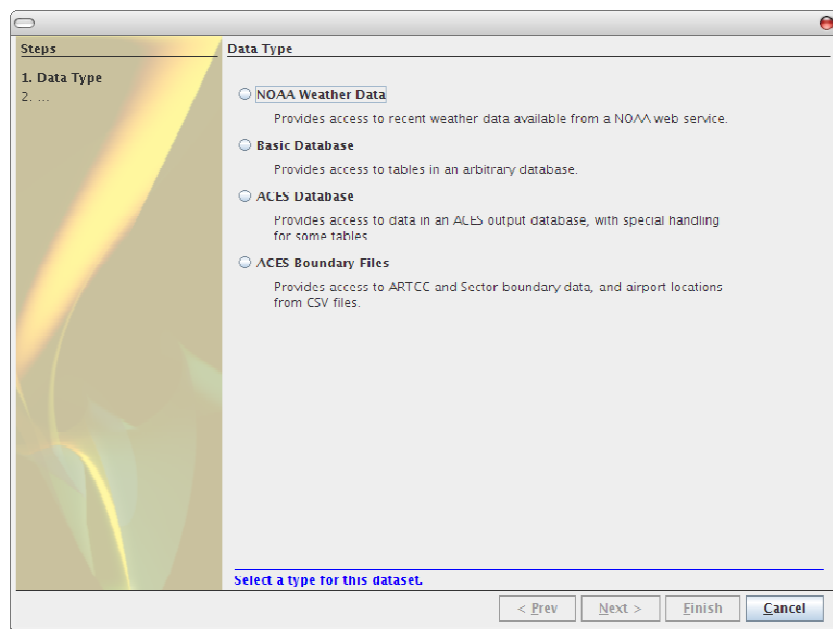


Figure 91: The new wizard user interface

The new Wizard (Figure 91) only allows users to select a single dataset extension to define parameters for. This means that datasets created with the new Wizard will only contain a single type of data. While this limits some of the functionality previously provided by the ACES Viewer, it also significantly improved usability.

4.4.5.4.2 CSV Dataset Definition

We improved the usability of the user interface component for importing comma-separated values (CSV) formatted data based on user feedback. The new user interface shows a preview of the first few lines of data in the file, and allows users to specify the data type and (optional) units for each column. We also improved the type inferencing capabilities of the CSV data subsystem in the ACES Viewer, to provide better default type selections for each data column, and to limit the available types to those that can be parsed from the data. An example of the new user interface is shown in Figure 92.

Steps

1. Data Type
2. Select CSV Files
3. Update Schema

Update Schema

Fill in the Column Names, Column Types, and Optional Column Units of Measure for the CSV file contained in /home/krisher/data/aces-input-tbl/CenterBoundary.csv

Name	Center	Low Boundary(lat/ion)	High Boundary(lat/ion)
	Type String	PolyData	PolyData
	Unit		String
ZAB	35.76666667/-111.8...	35.76666667/-111.8...	
ZAU	41.66666667/-93.46...	41.66666667/-93.46...	
ZBW	43.63333333/-76.79...	43.63333333/-76.79...	
ZDC	39.16666667/-80.4...	39.16666667/-80.4...	
ZDV	44.95833333/-103.1...	44.95833333/-103.1...	
ZFW	35.82916667/-100.0...	35.82916667/-100.0...	
ZHU	29.76666667/-102.5...	29.76666667/-102.5...	
ZID	39.86833333/-88.14...	40.0/-88.25 40.0/-8...	
ZJX	31.28055556/-87.4...	31.28055556/-87.4...	
ZKC	39.46666667/-98.8...	39.46666667/-98.8...	
ZLA	34.5/-123.25 35.533...	34.5/-123.25 35.533...	
ZLC	49.0/-114.6666667...	49.0/-114.6666667...	
ZMA	27.78333333/-85.0...	27.78333333/-85.0...	
ZME	36.01666667/-95.61...	36.01666667/-95.61...	
ZMP	49.0/-103.1666667...	49.0/-103.1666667...	
ZNY	42.72777778/-76.69...	42.72777778/-76.69...	
ZOA	40.98333333/-126.9...	40.98333333/-126.9...	
ZOB	43.5/-85.0 43.91666...	42.87083333/-82.46...	
ZSC	48.22222222/-128.0...	48.22222222/-128.0...	

< Prev Next > Finish Cancel

Figure 92: CSV data import wizard

4.4.5.4.3 Dataset Toolbox

The dataset panel that displays a listing of loaded datasets and data tables within each dataset has been significantly improved. Previously, the user interface for managing datasets was spread between several menu items, toolbar buttons, and other panels. To improve usability, we collapsed all of this functionality into a single location in the interface. The new dataset panel contains hyperlink style buttons that allow users to load data or define a new dataset. The ability to unload datasets is now accessed directly from the dataset listing, via 'X' buttons next to each removable dataset. The new panel is illustrated in Figure 93.

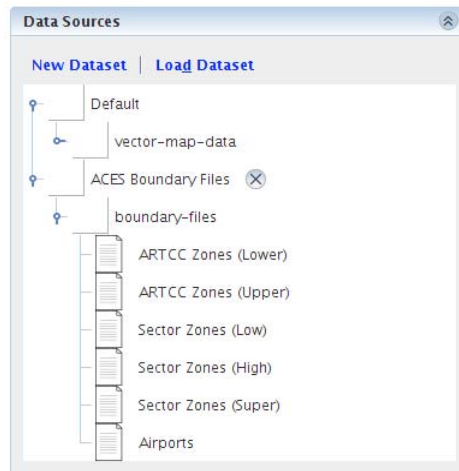


Figure 93: Dataset management panel

4.4.5.4.4 Dataset Resolution

We re-designed parts of the Dataset API to allow saved visualizations to be more easily applied to data sources other than those that were present when the visualization was persisted. The changes allow data sources to be referenced by Schema rather than by identifier in saved visualizations. This allows any data source with a compatible Schema to be used as a replacement when loading a visualization. This provides more flexibility than the previous implementation where only data sources from a similar media could be used. For example, we can now save a visualization containing data sources representing relational database tables, and later load the visualization replacing those data sources with CSV file data.

The Dataset Resolver Dialog was developed to expose the new capabilities of the Dataset API. It is displayed when loading a saved visualization to give users an opportunity to select the data sources that should drive the visualization from a list of those that are determined to be compatible based on the Schema of each source. The user interface is shown in Figure 94.

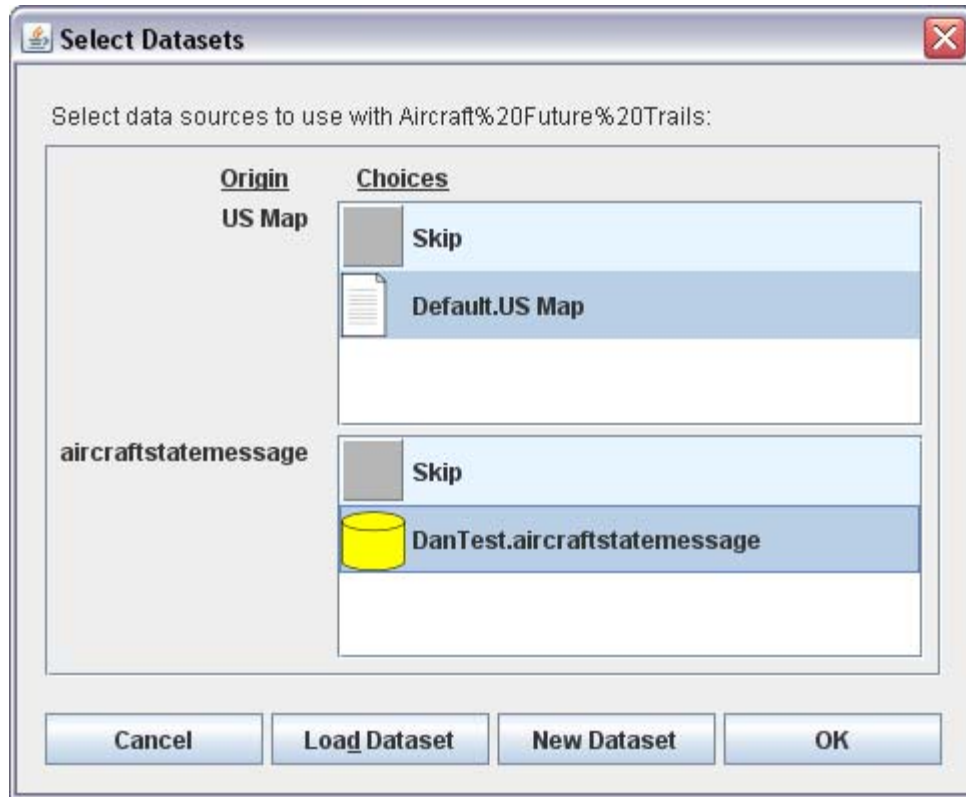


Figure 94: The Dataset Resolver Dialog

4.4.5.5 Property Sheet

The Property Sheet user interface component, which allows users to manipulate the settings for each of the components that make up a visualization, has been extensively refactored to improve consistency, performance, and flexibility. The work on this change was split between the UDOP project and the ACES Viewer since it benefits both projects. Major changes include:

- Consistency. All property editors now occupy a single line, and can have optional popup or dialog based editors when a more complex user interface is necessary.
- Simplicity of Implementation. The property sheet and property editor API was previously based on a built-in Java API for working with Java Beans. This API had a lot of extra functionality that was not used by the ACES Viewer, but made implementation of new editor components more complex. We developed a new API to address these problems.
- Flexibility. We designed improved support for dynamic property changes, embedded properties, and other functionality that simplifies development of editors, and provides more appropriate user interface components for the visualization component being edited.

An example of the new user interface is shown in Figure 95.

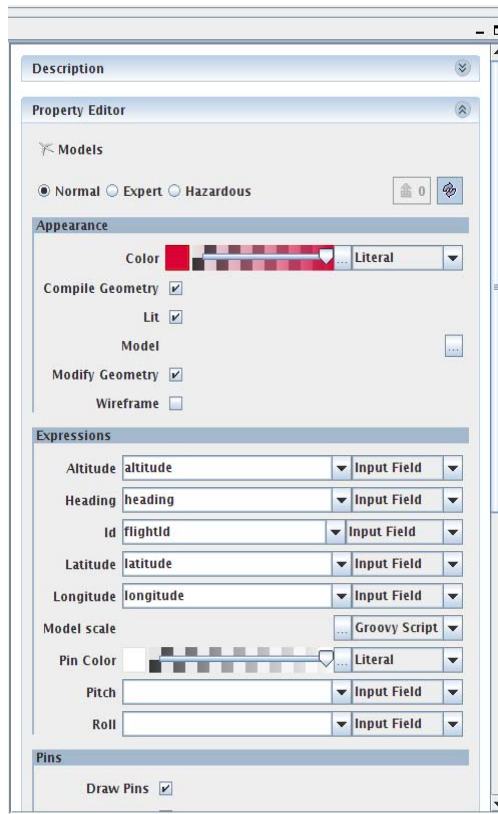


Figure 95: New Property Sheet User Interface

4.4.5.5.1 Expression Editor

We refactored the Expression Editor UI component to use pluggable ExpressionFactory component to define how to create and edit each of the available expression types. This simplifies the changes necessary to add support for new expression types, and for modifying the user interface components for manipulating existing expression types.

4.4.5.5.2 Schema Editor

We developed a user interface component to define and manipulate Schema objects. The editor (Figure 96) allows users to enter column names, types and units. Columns can be added or removed using the appropriate buttons. Column order can also be changed by dragging the row in the editor interface corresponding to the column to be moved (Figure 97).

<input type="checkbox"/>	name	String	N/A	×
<input type="checkbox"/>	data	PolyData	N/A	×
<input type="checkbox"/>	centerLatitude	Number	Unit	×
<input type="checkbox"/>	centerLongitude	Number	Unit	×

Add Field

Figure 96: The Schema Editor

<input type="checkbox"/>	name	String	N/A	×
<input type="checkbox"/>	centerLatitude	Number	Unit	×
<input checked="" type="checkbox"/>	data	PolyData	N/A	×
<input type="checkbox"/>	centerLongitude	Number	Unit	×

Add Field

Figure 97: Reordering Columns in the Schema Editor

4.4.5.5.3 World Data Configuration Editor

The Terrain Renderer in the ACES Viewer (based on the World element in JView) requires a variety of paths to be specified by the user in order to locate the relevant elevation and imagery data. We developed a wizard style user interface component to specify the necessary information (Figure 98).

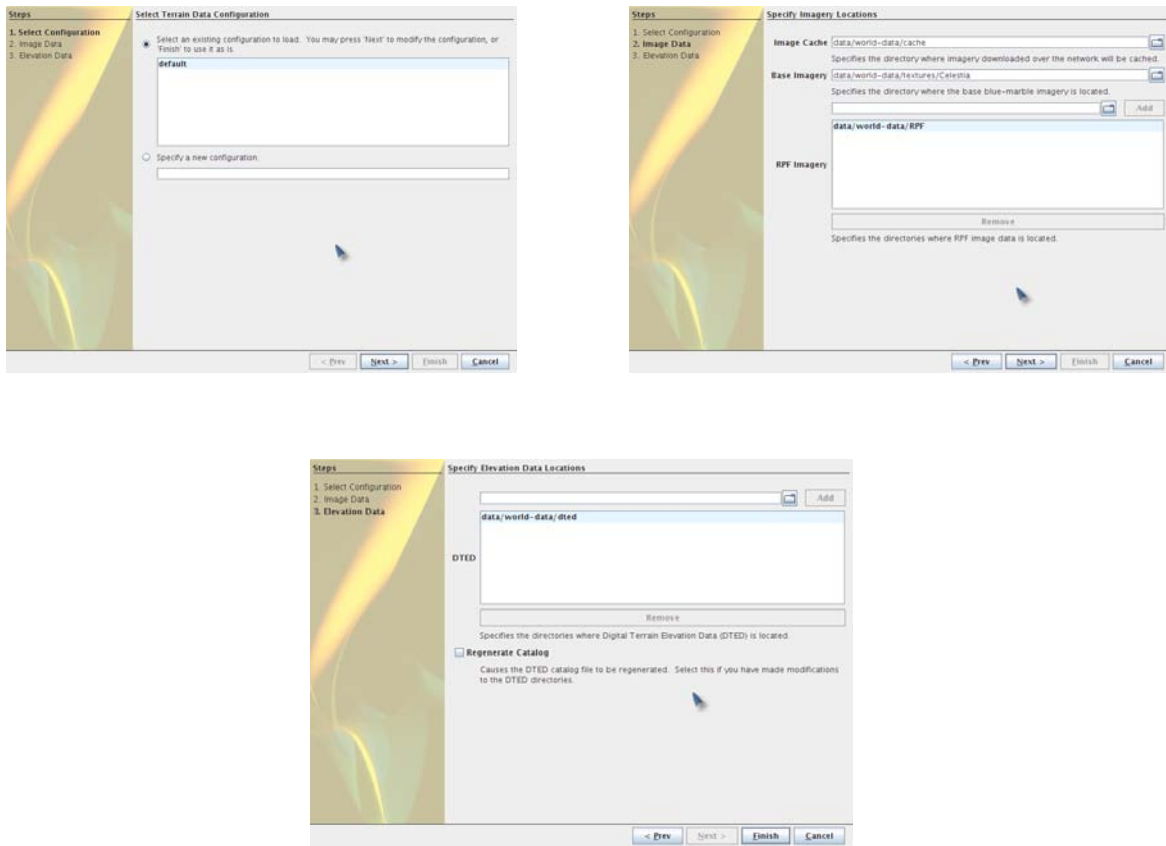


Figure 98: World Data Configuration Wizard

4.4.5.6 Visualization Composition Graph

4.4.5.6.1 Save to CSV

Previously, we had added a system to save ACES Viewer data represented in Tables to disk in CSV format. We have now added a method for users to access this functionality via the context menu in the Composition Graph user interface component. Users may now select and save individual tables represented in the graph to disk.

4.4.5.6.2 Cut/Copy/Paste

We have added support for copying visualization components present in the active visualization container, and pasting a copy back into the same container or another. When pasting, a new visualization component is created with the same type as the one that was copied. All of the values of user editable properties are then copied from the source component to the pasted component. This produces an identical visualization component, with the exact same configuration as the copied component.

4.4.5.6.3 Drag and Drop Composition

We have improved the drag and drop interface used with the visualization composition graph. Drag and drop support in this component allows users to easily add components from a toolbox to an active visualization. We have improved the usability of this system by providing active visual feedback in the composition graph that indicates the container that will be modified if the dragged component is dropped at the mouse pointer location. It is also now possible to drop components onto other components in the composition graph to place them in the same container.

4.4.5.6.4 Automatic Layout of Graph Nodes

The original Visualization Composition Graph relied on manual placement of the components in the graph for layout. We experimented with several graph layout algorithms including various tree layouts, force directed layouts, and hierarchical layouts. We found that the hierarchical layouts produced consistently good looking graphs, and developed a custom implementation for the ACES Viewer. The hierarchical layout also provides the added benefit of being deterministic (the layout produces identical results given the same graph components in the same order), unlike other algorithms such as the force directed model.

The algorithm is based on a hierarchical layout model, where the Renderers are considered roots of the graph, and all other components are laid out based on their graph distance from a Renderer. The algorithm works as follows:

1. All graph nodes are ranked by the longest graph distance to a Renderer identified using a Breadth First Search (BFS) (this gives the 'longest-shortest distance to a Renderer'). Each node is assigned a column in a grid based on this ranking
2. Rows are assigned to each node using a pseudo-topological sort. Nodes in each column are ordered by the order they appear in a post-order Depth-First traversal from nodes that have no left-connected neighbors from step 1. Rows are assigned based on this ordering, the average row value of neighbors, and any grouping constraints (e.g. Renderers in the same scene should appear next to each other).
3. X and Y coordinates are assigned to each node based on column and row assignments, and the widths/heights of other nodes in the same row/column.
4. Nodes representing groups (Scenes) are placed to surround the visualization components they contain.

Figure 99 shows several results of this layout algorithm applied to ACES Viewer visualizations.

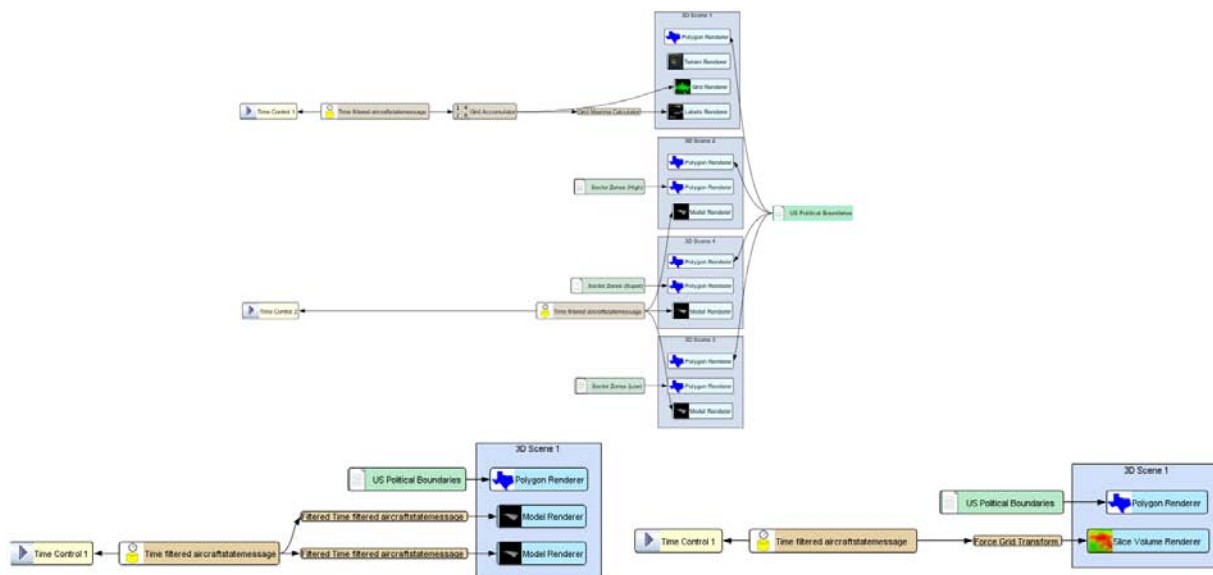


Figure 99: Several graph layouts produced by the new Hierarchical layout algorithm

4.4.5.6.5 Other Composition Graph Enhancements

We made several additional improvements to the visualization composition graph appearance and functionality. Changes include better icons for visualization components, improved support for components with multiple inputs and outputs, better component layout, and improvements in the algorithms to determine which components may be linked together. A sample graph illustrating some of the changes is shown in Figure 100.

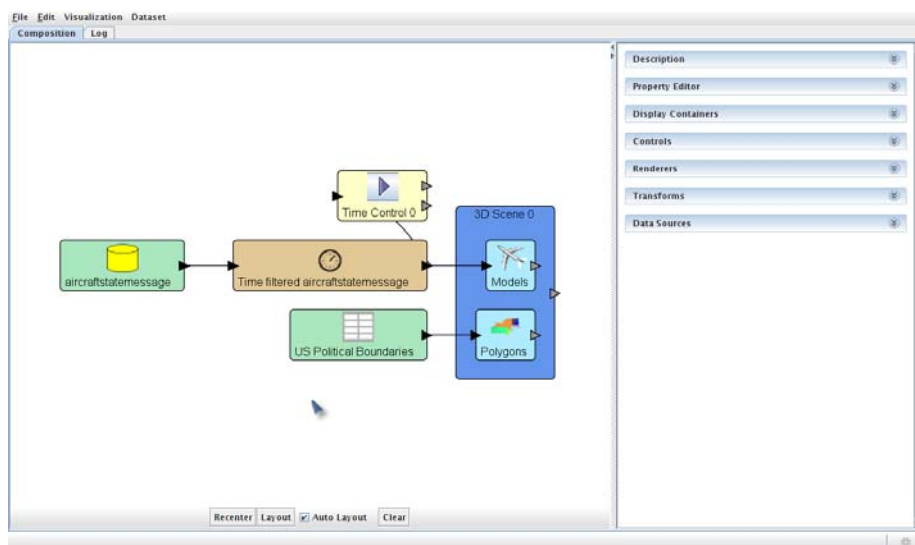


Figure 100: Visualization Composition Graph

4.4.5.7 Visualization Composition Toolbox

The visualization component toolboxes used with the visualization composition graph have been enhanced using JView's ImageList component. This component makes more efficient use of screen space, allowing more visualization components to be visible in the toolboxes, as shown in Figure 101.

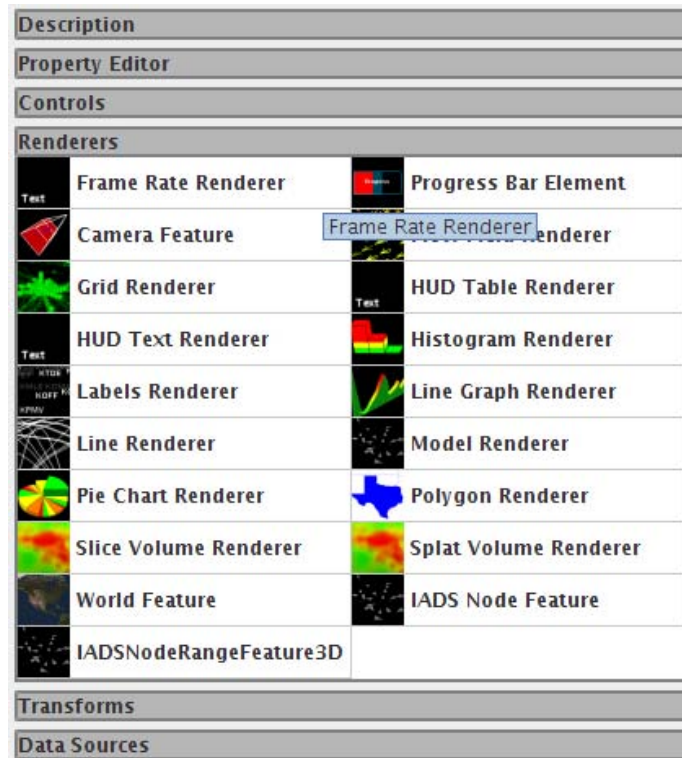


Figure 101: An Early Version of the Renderers Toolbox, Displayed Using JView's ImageList

Due to inconsistencies in the appearance of JView's JCollapsePanel component across different platforms, we replaced them with the JXTaskPane component from the Sun endorsed SwingX library. The functionality of the JXTaskPane is similar to the JCollapsePanel, but it respects the look and feel, and human interface guidelines of the major platforms supported by Java. Figure 102 shows the task panes as they currently exist in the ACES Viewer.

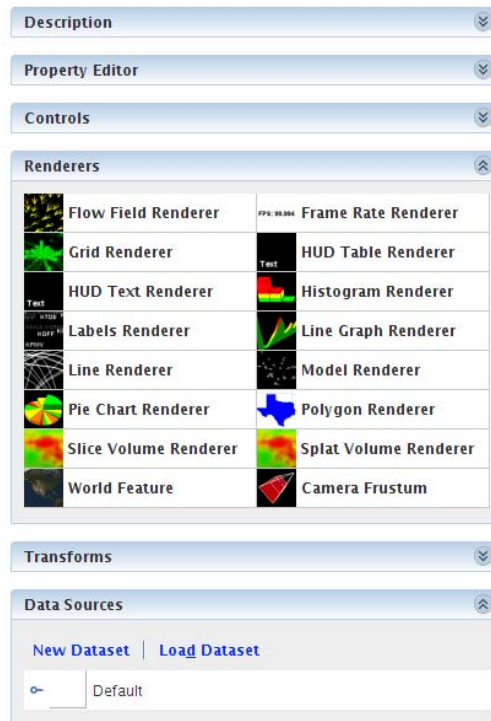


Figure 102: JXTaskPanes that replace the previous JCollapsePanel component

4.4.5.8 Movie Capture

To improve usability, we made some changes to the movie capture user interface. Previously, users initiate movie capture through a menu item. This caused a new window, the *capture window* to be displayed. Users could then drag scenes to the capture window to enlist them in the recording process. The new interface is initiated the same way, through a menu item. The difference is that the movie capture controls are added directly to the window from which the user selects the menu item to begin recording. Several buttons and labels were also reorganized to make their functionality more obvious. The new interface is shown in Figure 103.

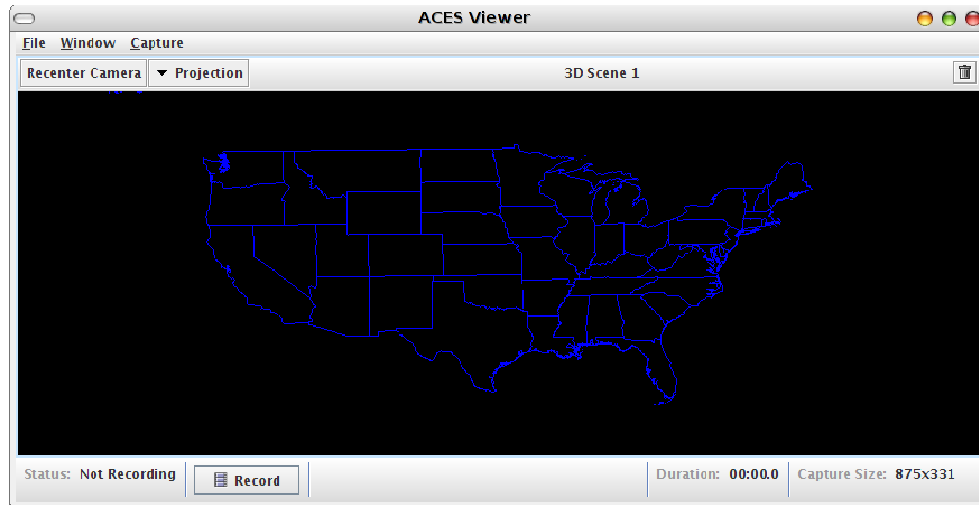


Figure 103: The new movie capture user interface

4.4.5.9 Development and Debugging Tools

We developed a user interface component to display a list of all content-change events issued by active Tables (data sources and Transforms) in the application (Figure 104). This component is extremely useful for testing and debugging when developing new Transforms or dynamic data sources (e.g. connected to live data streams). Each time an event is issued from a Table, a row is added to the user interface describing the source of the event, the nature of the event (how many rows were added/removed/changed, and whether the schema of the issuing table has changed), and the number of rows present in the issuing Table after the change that caused the event.

Name	Schema	All	Added	Removed	Changed	Tuple Count
Polygon Centroid Location Transform		X				0
HighSector.csv with Aggregate		X				0
Aggregate		X				12
Time filtered aircraftstatemessage		X				67
Polygon Centroid Location Transform		X				1
HighSector.csv with Aggregate		X				1
Aggregate		X				19
Time filtered aircraftstatemessage		X				90
Polygon Centroid Location Transform		X				1
HighSector.csv with Aggregate		X				1
Aggregate		X				29
Time filtered aircraftstatemessage		X				97
Polygon Centroid Location Transform		X				1
HighSector.csv with Aggregate		X				1
Aggregate		X				41
Time filtered aircraftstatemessage		X				116
Polygon Centroid Location Transform		X				1
HighSector.csv with Aggregate		X				1
Aggregate		X				41
Time filtered aircraftstatemessage		X				117
Polygon Centroid Location Transform		X				1
HighSector.csv with Aggregate		X				1
Aggregate		X				45
Time filtered aircraftstatemessage		X				131
Polygon Centroid Location Transform		X				3

stop clear

Figure 104: Table Event Viewer User Interface Component

4.4.6 Release Management

4.4.6.1 Native Packaging

We incorporated utilities into the ACES Viewer build system that allows automated packaging of the application for release. These utilities include the ability to generate native application launchers for Mac and Windows platforms.

4.4.6.1.1 Mac

Using an open source third-party plug-in (JarBundler) for the ANT build tool, we added support for generation of a native Mac OSX application bundle for the ACES Viewer. This allows the ACES Viewer to be launched by double clicking an icon, like any other native application. The bundle can now be created from the latest sources by simply executing 'ant mac-bundle' in the ACES Viewer source directory.

4.4.6.1.2 Windows

Using Launch4J we created a native windows launcher for the ACES Viewer. The launcher functions like most other .exe executables on Windows, allowing users to double click an icon to launch the application.

4.4.6.2 Releases

4.4.6.2.1 ACES Viewer 1.0

Early in this contract we produced the first stable release of the ACES Viewer with Interactive Visualization in 4D (IV4D). IV4D is an ACES Viewer plug-in designed to provide a data-centric user interface and visualization configuration layer to simplify the process of creating an initial visualization for NASA researchers. Besides reviewing, testing, and improving the application code, we also worked with the Information Handling Branch (AFRL/RIEB) to prepare the distribution media, set up a customer support mailbox, and ship the software to users who had requested it. We also coordinated with the contractor developing the IV4D extension to the ACES Viewer for simultaneous release. The ACES Viewer 1.0 release was completed and shipped on June 5, 2008.

4.4.6.2.2 ACES Viewer 1.1

We released version 1.1 of the ACES Viewer and the IV4D extension to the NextGen researchers and other interested parties in August 2008. This was primarily a bugfix/maintenance release.

4.4.6.2.3 ACES Viewer 1.5

ACES Viewer 1.5 was released in May 2009. This release added some new features, and addressed several usability issues present in earlier versions.

4.4.6.2.4 ACES Viewer 2.0

Version 2.0 of the ACES Viewer was released in January 2010. This version included major revisions to the ACES Viewer API, infrastructure and user interface.

4.4.6.2.5 ACES Viewer 3.0

We released version 3.0 of the ACES Viewer to NASA in June 2010, which included a huge number of improvements for performance, memory efficiency, and usability over the previous 2.0 release. The release also included a variety of changes in support of the development of IV4D.

4.5 Characterization of the UAV Network Environment (CUNE) Viewer

The Characterization of the UAV Network Environment (CUNE) team needed an application to view time keyed GPS data on the world. The CUNE team attempted to use Google® Earth to display this information but they quickly found limitations in what they could show. As a result a JView based CUNE Viewer application was developed to show the GPS location sequences, with additional information encoded in the visual representation. The view (Figure 105) shows the GPS line with an RC plane model (representing the vehicle that is used to collect the data). The color of the line represents the GPS signal quality at the corresponding location/time. A data playback control is located on the bottom right and will help the user navigate through temporal data. An information panel is located at the top right showing the current signal stats (Figure 105).

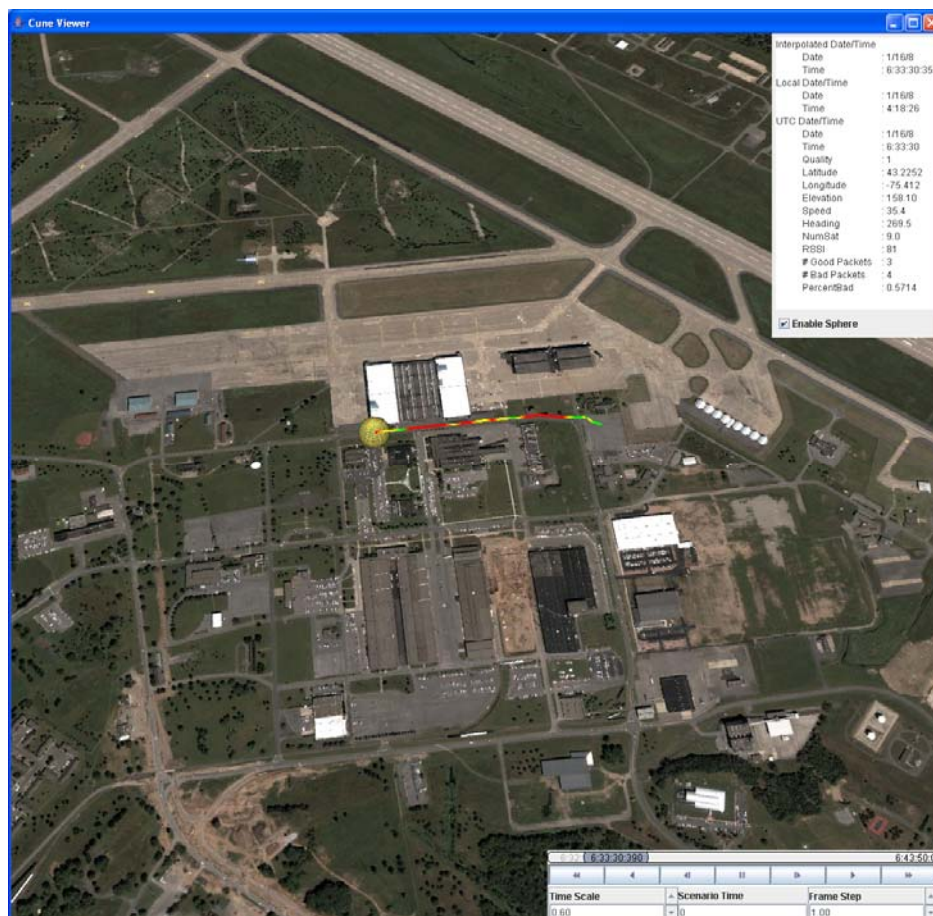


Figure 105: An Early Version of the CUNE Viewer

The CUNE Viewer can connect to either a live connection where information is retrieved through a TCP connection to server software developed for the CUNE project, or from recorded data. Upon loading a file or initiation of a live connection, users can play through the data, with a number of different visualization options. Some rendering options include history trails, antenna plots, and a communication line. The history trail of the UAV is a line segment showing where the UAV has been. It is colored based on the number of good packet vs. bad packets the destination received at a particular time. We have incorporated the antenna plot element originally developed for the Antenna Pattern Analysis Tool Set. This places a sphere like object around the UAV showing the signal strength characteristics of the antenna. There are four different antenna plots: lines, points, pincushion, and surface as seen in Figure 106. The communication line is an animated line that shows the direction of information. Figure 107 shows the CUNE viewer where the source is the model aircraft, the destination is the truck, the antenna plot is the translucent sphere like object around the aircraft and the communication line is the yellow line from the truck to the aircraft.

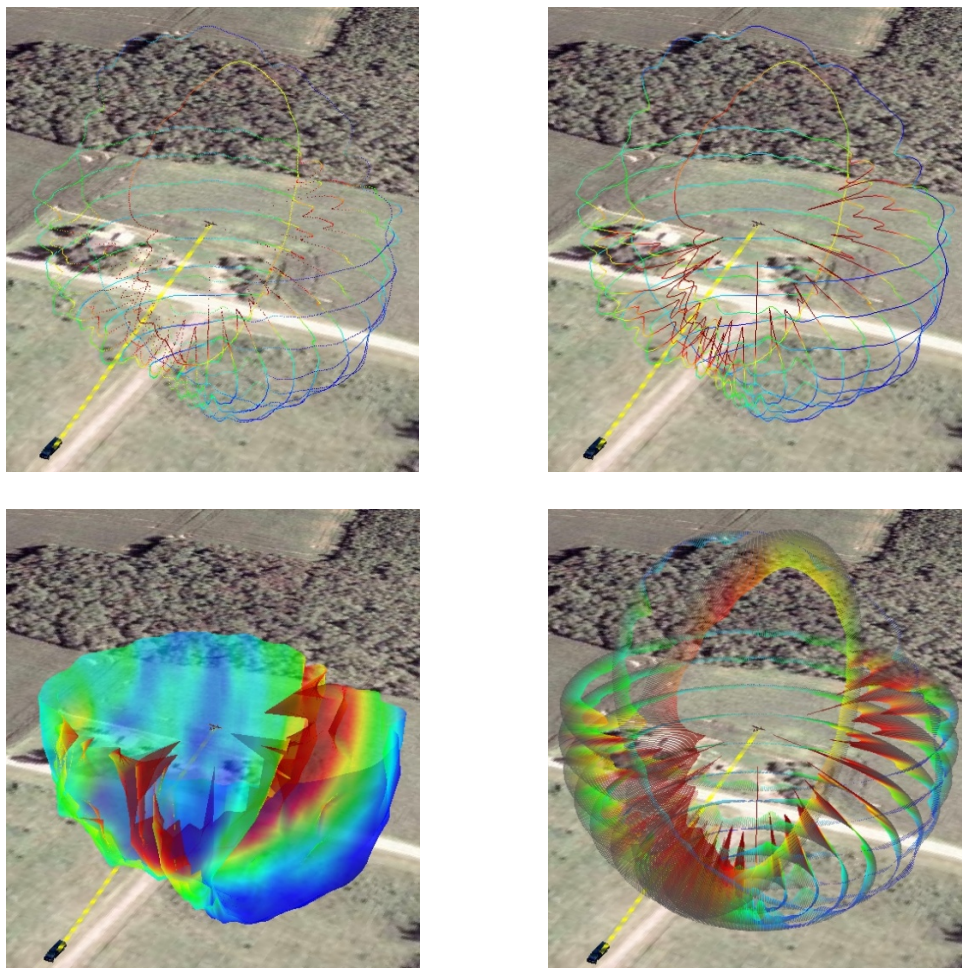


Figure 106: Antenna Plots (Point, Line, Surface, Pincushion)

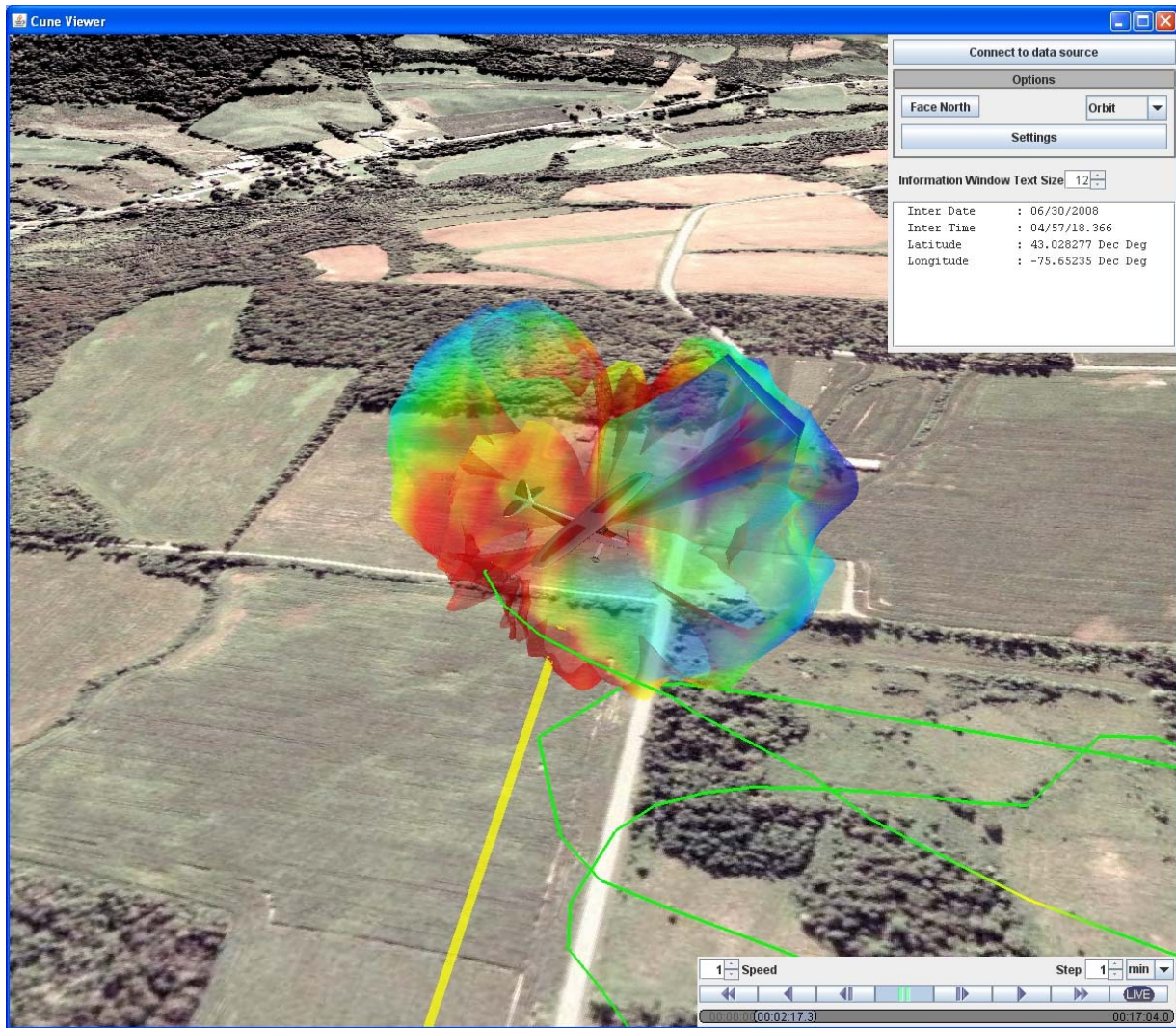


Figure 107: Cune Viewer

To slim down the CUNE viewer interface we added a settings panel. The setting panel allows all the options to be in an organized fashion without all the clutter in the main GUI. This separate window provides more space for options compared to the earlier versions of the user interface. Figure 108 shows the settings window that the user can pop up by pressing the settings button.

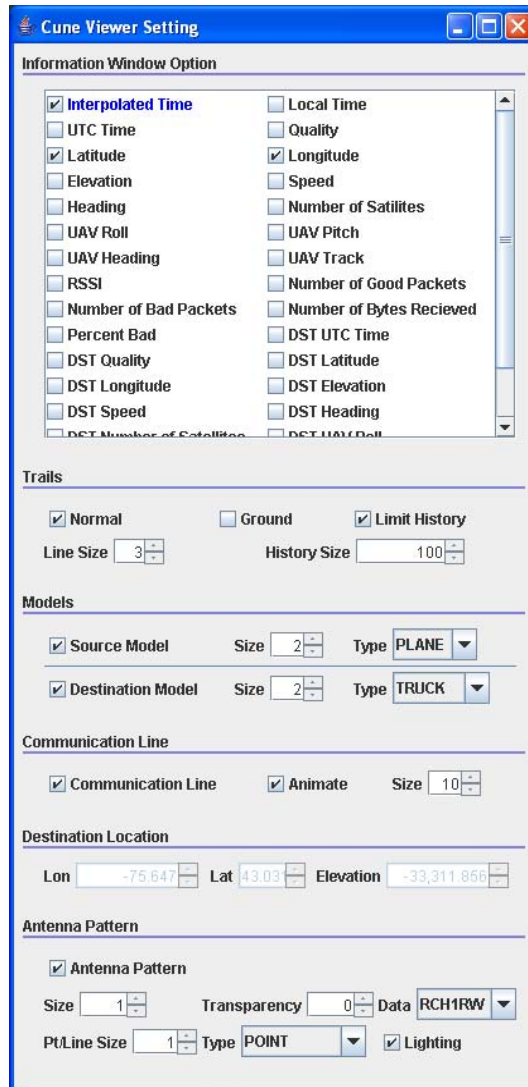


Figure 108: Settings Window

A configuration file and manager was created for the CUNE viewer so that users can tweak settings such as format and default settings. Upon starting the CUNE viewer the format file will be read and set to the appropriate settings. Figure 109 shows an example of what a CUNE configuration file would look like. On the left there are the properties that can be changed and on the right separated with a '::' is the value of the property.


```

WORLD LEVEL OF DETAIL      :: HIGH
WORLD LOAD FROM URL       :: FALSE
TRAIL SELECTED            :: TRUE
GROUND TRAIL SELECTED     :: FALSE
DATE FORMAT               :: MM/dd/yyyy
TIME FORMAT               :: kk:mm:ss.S
LONGITUDE FORMAT          :: ###.#####°
LATITUDE FORMAT           :: ' ##.#####°'
ELEVATION FORMAT          :: ##.##### m
PITCH FORMAT              :: ##.#°
ROLL FORMAT               :: ##.#°
HEADING FORMAT            :: ##.#°
TRACK FORMAT              :: ##.#####°
SPEED FORMAT              :: ##.##### km\h
PACKET FORMAT             :: ###
BYTES FORMAT              :: ##### bytes
QUALITY FORMAT            :: ###
NUM SAT FORMAT            :: ###
RSSI FORMAT               :: #### db
ALIGN DECIMAL             :: true
ALIGN NUMBER TO DECIMALS  :: true

```

Figure 109: CUNE Configuration File

The CUNE viewer can show many communication links to a particular source and also that source trial can be colored by any parameter that the user wishes. In the panel it shows all the connection points represented as bars, each bar has a zoomed bar and overview bar. The overview bar which is the top bar shows the whole communication to the source throughout the simulation. It is colored by the user and is defined in the upper left hand corner. In the picture below (Figure 110) these simulations are colored by Received Signal Strength Indicator (RSSI) values and red is 100 and 40 is green. This panel allows users to define what the line represents. For instance, a particular communication line would represent how many bytes have been received. Additionally, in this panel users can change the color mapping to values. For example, 10 bytes received is colored red and 200 bytes received is colored blue. They can also choose which communication line to view by selecting on the title.

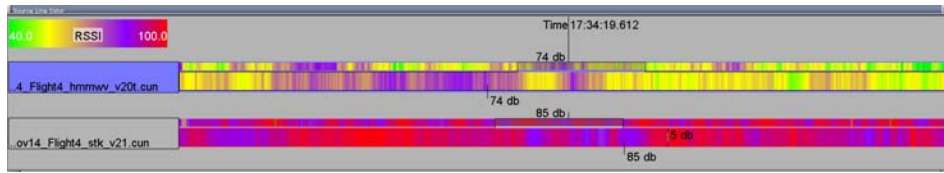


Figure 110: Communications Signal Panel

Another feature that was added to the CUNE viewer was a popup charting panel to show the charts of the information currently loaded. Below, Figure 111 shows the chart panel in the CUNE viewer.

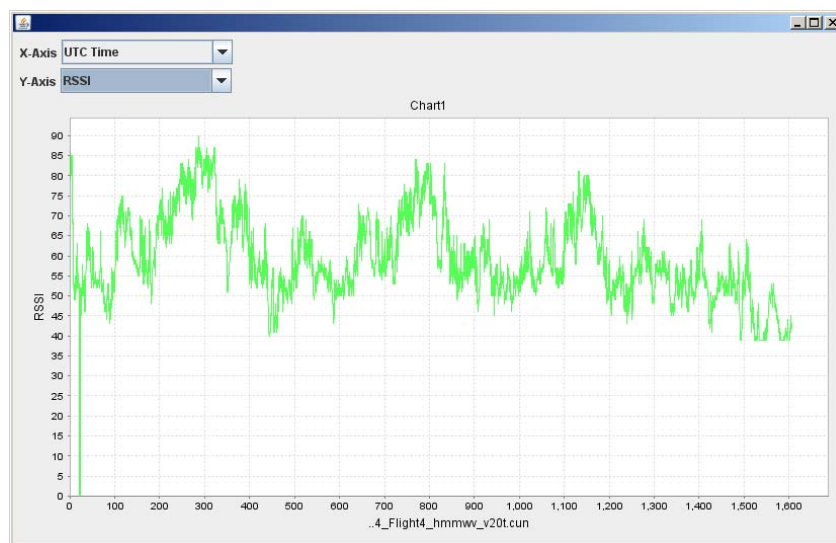


Figure 111: The Chart Panel in the CUNE Viewer

Upon request from the CUNE team, a movie tool was added to the CUNE viewer to have the ability to create movies of the visualization. Movies demonstrate the application or simulation as well as creating a distributable media where anyone can view it without knowing or having the application.

We designed the CUNE viewer to be viewed on many types of screens. Most application developers design their components to fit a standard resolution computer. Unfortunately, when the resolution increases the graphical components get so small that a normal user has difficulty using the application. The CUNE viewer was designed for three display scales, small, medium and large. The user interface is scaled to the size selected so the application can be used on a variety of different devices with different size and resolution characteristics. The illustrations below are the CUNE Viewer on two different types of resolution. On the top shows the CUNE Viewer set to small (Figure 112) and on the bottom (Figure 113) it is set to large.

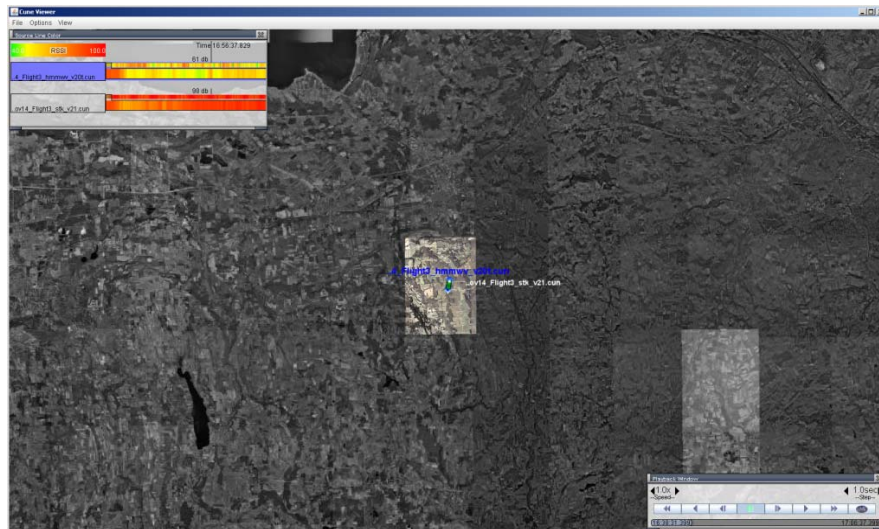


Figure 112: CUNE Viewer on Small

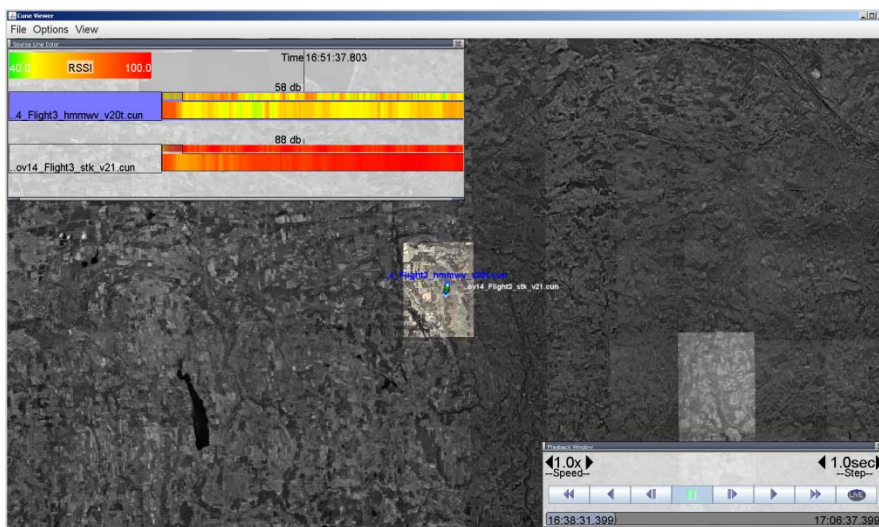


Figure 113: CUNE Viewer on Large

A close-up view of the UAV antenna pattern enables users to see the intersection of the communication line and the antenna pattern that was collected at the lab. This intersection indicates the value and location of the intersection, which are both useful for future analysis. Figure 114 shows a close up of a model with communications lines extruded from it. The numbers associated with the communication lines shows the value of the antenna pattern at that point.

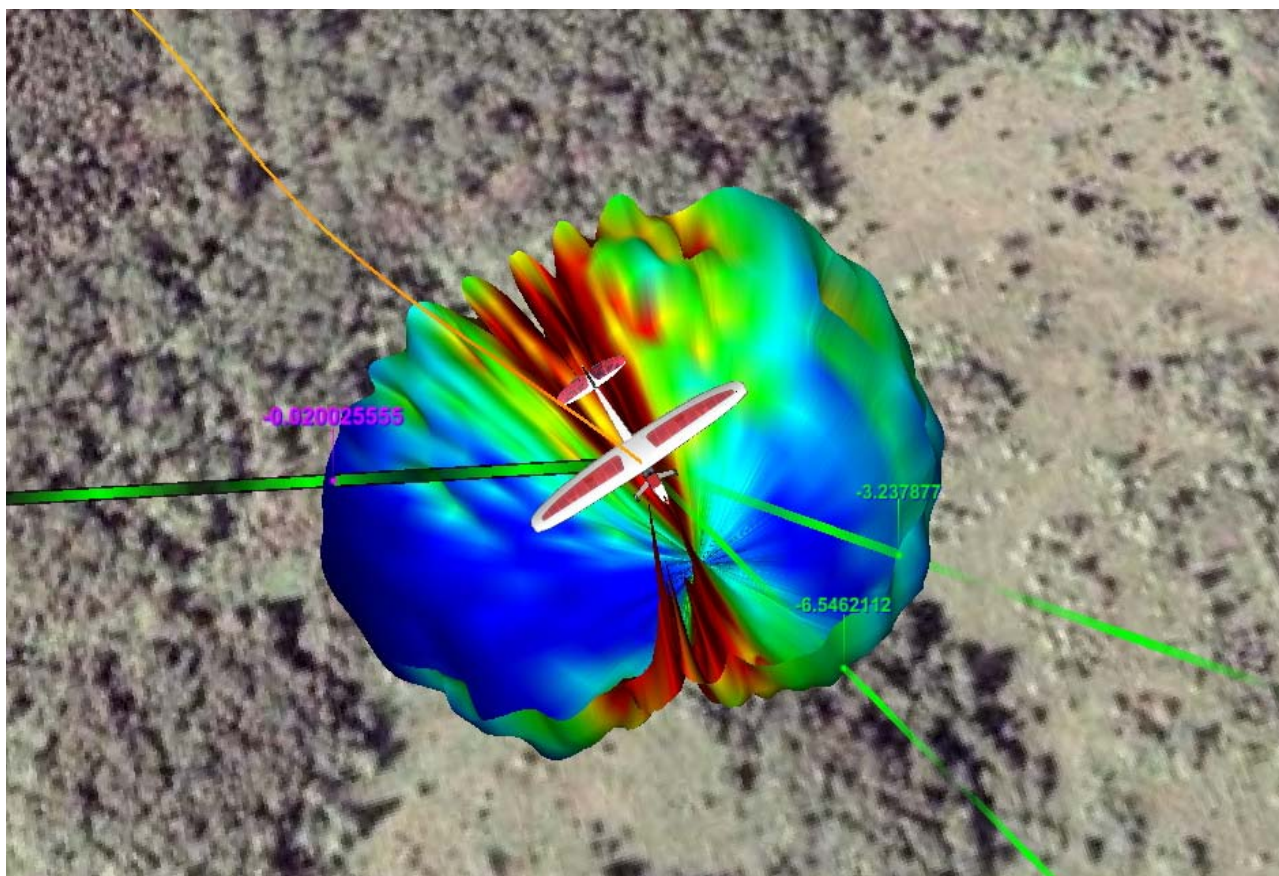


Figure 114: CUNE Viewer

4.6 JWeather

JWeather is an effort to develop a visualization of weather and its impact to operations such that decision makers are able to perceive their situation with new clarity. This effort leverages existing JView and ACES Viewer capabilities as well as extending support to include operational weather distribution and visualization requirements.

4.6.1 Background Research on Contemporary Weather Visualization Techniques

We researched various methods online of visualizing weather and documented the links on the AFRL wiki (https://twiki.tbm.rl.af.mil/wiki/index.php/Weather_Visualization). We also discussed techniques with weather expert, Bob Farrell of the Cyber Command and Control Systems Branch (AFRL/RISF), and received some insight from him on the various weather sources and how to interpret some of the data. We determined from these discussions that wind and cloud coverage would be two forms of weather that would be desirable to visualize.

4.6.2 Access to Real-Time Weather Data

We collected online documentation on the Joint METOC Broker Language (JMBL). There are different versions of JMBL for Jet, WDA and JWIS. We also developed software to access the data based on the specifications.

After receiving an account for the local JWIS server we wrote a SOAP interface to retrieve data from it. The SOAP requests were initially hardcoded to retrieve a fixed set of gridded data, however we eventually rewrote the initial data access system to remove the hardcoded parameters and allow flexible requests. Unfortunately there is no way to request a catalog of what is available on the server, however the Jet server (which was not available at the time of the initial development effort) should have it.

After developing the initial data access components for the local JWIS server, we modified the JMBLWeather class to retrieve data from the AFWA server. This involved implementation of support for SSL SOAP transfers to be able to talk to the AFWA server.

To improve the responsiveness of visualizations of weather data, we developed a caching system that temporarily stores weather data fetched from a remote server locally.

4.6.3 JView Based Visualizations for 4D Weather Data

We developed an initial wind pattern visualization from the JWIS server that uses the windU and windV 2D gridded data sets to determine wind speed and direction then display them on the map using the Flow Field Renderer previously developed for the ACES Viewer.

We determined a majority of the data on the JWIS is gridded in a Lambert Conformal which was not initially supported in Flow Field Render. CACI has made the changes to allow data in projections other than WGS84. Figure 115 and Figure 116 show examples of Lambert Conformal input data displayed on a WGS84 and Flat Earth World.

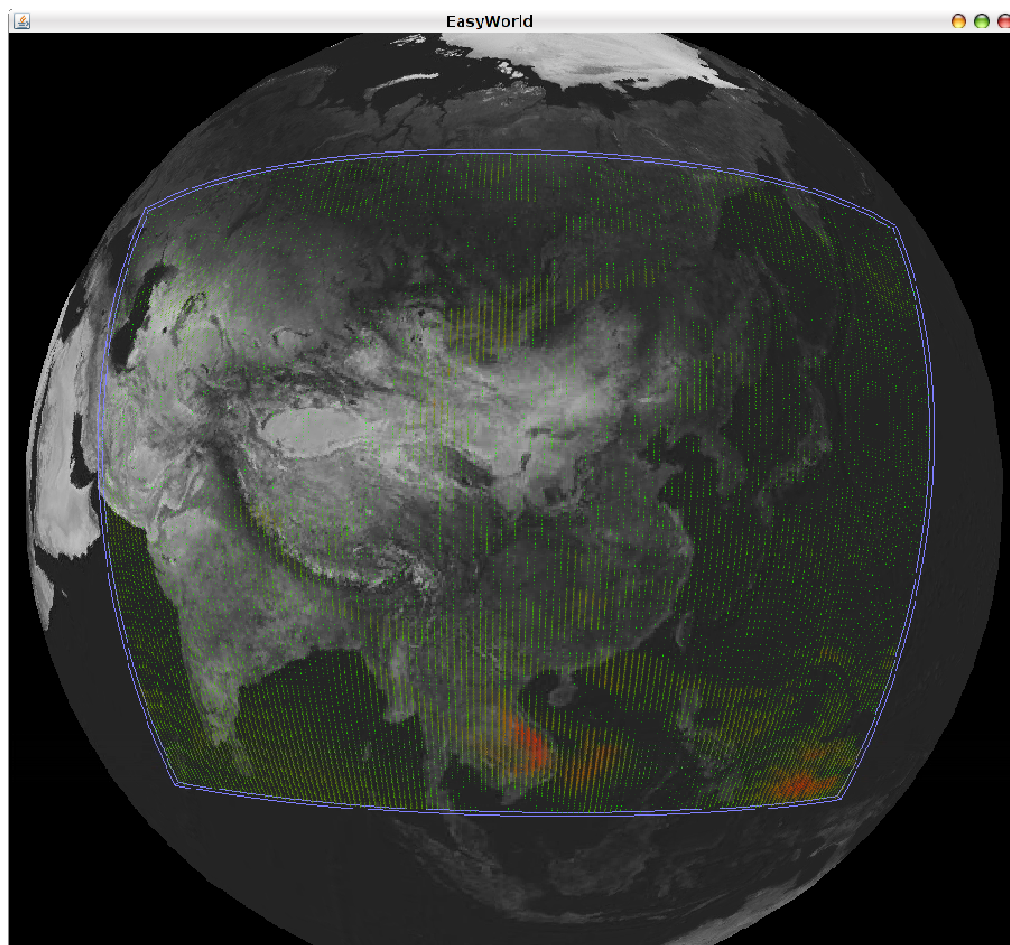


Figure 115: Weather Data on WGS84 World

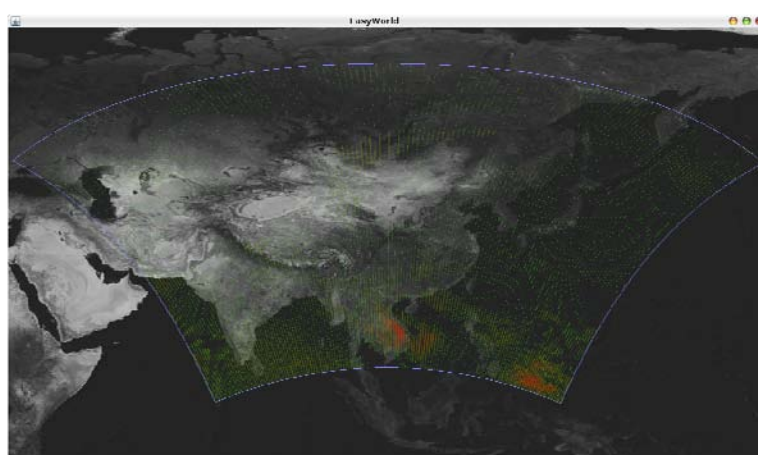


Figure 116: Weather Data on Flat World

We developed a kd-tree implementation that minimally supports fast range and nearest neighbor searches and conforms to the Java Map interface. It also supports “hinting” which allows specifying bounds the data will be in which aids in determining the partitioning of the tree. This was used to develop a new cloud visualization that uses the kd-tree to store and retrieve the cloud data covering the whole world (about 6MB per forecasted time) in an efficient tree that allows quick access to the multi-resolution cloud data for any given area (Figure 117).

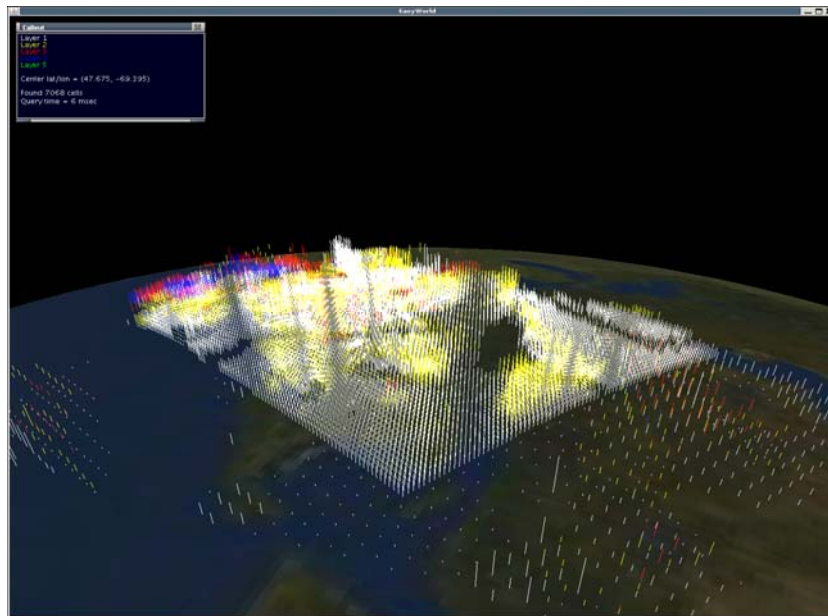


Figure 117: JWeather Cloud Visualization

We developed an isosurface renderer that rendered layered relative humidity (Figure 118).

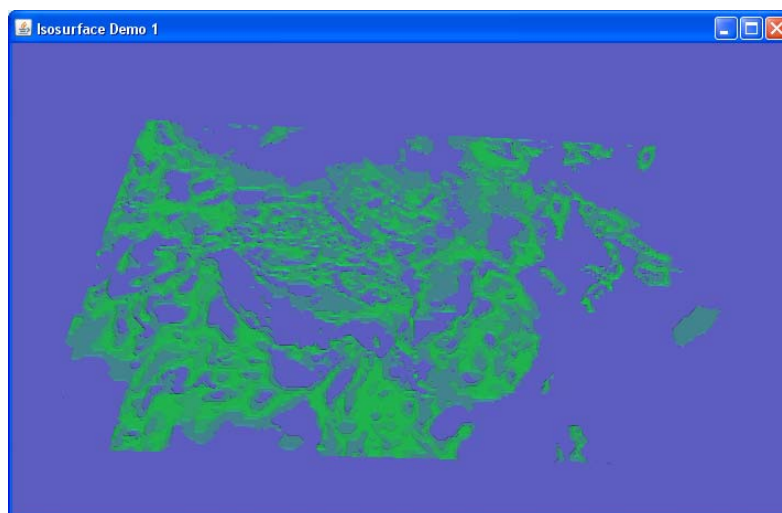


Figure 118: JWeather Isosurface Renderer

In order to develop improved visualization support for cloud data, we developed the necessary data structures to hold the Diagnostic Cloud Forecast Model (DCF) data:

- Fractional layer cloud amounts for up to four levels in the atmosphere
- Layer cloud base heights for up to four levels in the atmosphere
- Layer cloud top heights for up to four levels in the atmosphere
- Layer cloud types for up to four levels in the atmosphere
- Total fractional cloud amount for the entire atmosphere.
- Total fractional cloud amount for the entire atmosphere.

After the data model was developed, we started visualizing the DCF data on the JView World (Figure 119).

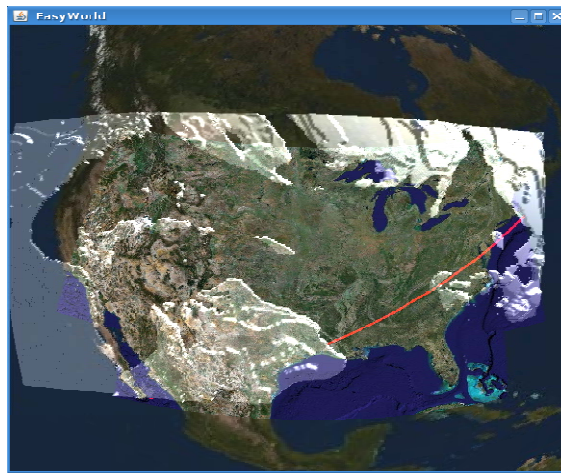


Figure 119: JView Visualization of DCF Data

We developed a new weather visualization geared toward avoidance of weather along a flight path. The current implementation colors a flight path (currently limited to a single line) based on the interaction an aircraft flying on that path with the 4D weather (Figure 120). This required developing a way of storing a vast amount of data in memory to hold the 3D weather over time (giving 4D weather). Thought has been given to moving this data store to a remote server since only small areas of weather data are required at a time, however it needs to be accessed quickly, which excluded the use of the AFWA server or local JWIS server.

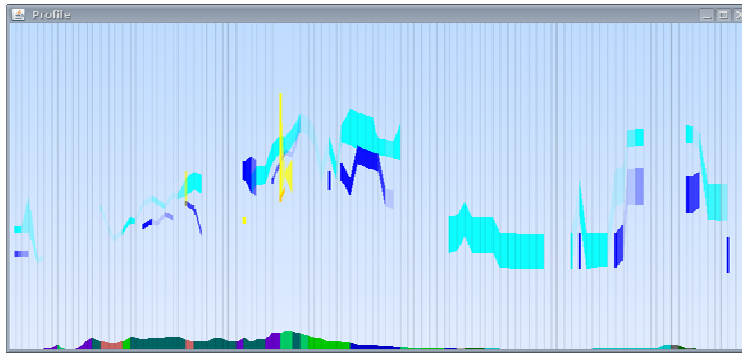


Figure 120: Visualization of Cloud Profile Along a Flight Path

We developed a multi source 3D cloud renderer using AFWA satellite imagery as the underlying texture on the world. The GRIB binary data was used to generate 3D geometric cloud cover over the satellite imagery. This was added to the weather test program, and gives us the capability to show the radar cloud imagery for the same time interval as the binary 3D data overlaid on the earth (Figure 121). This was done as a verification test of both the display and retrieval of the appropriate cloud forecast, as well as a test of AFWA's cloud forecast.

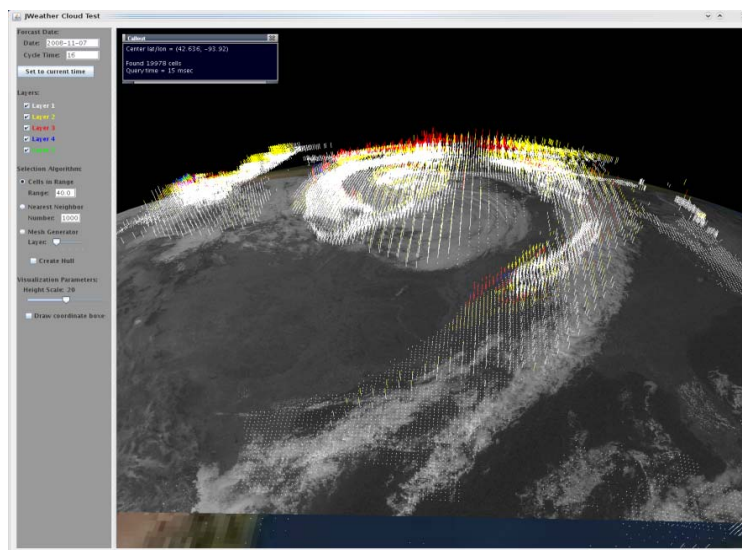


Figure 121: Cloud data from AFWA over the satellite imagery for the same forecast time

4.6.4 User Interface

We developed a graphical user interface (Figure 122) to control the features of the kd-tree and the visualization. Now it is possible to select the forecast date, visible layers, height scale and the selection algorithm between cells in range, nearest neighbor and mesh generator as well as the parameters that affect their output.

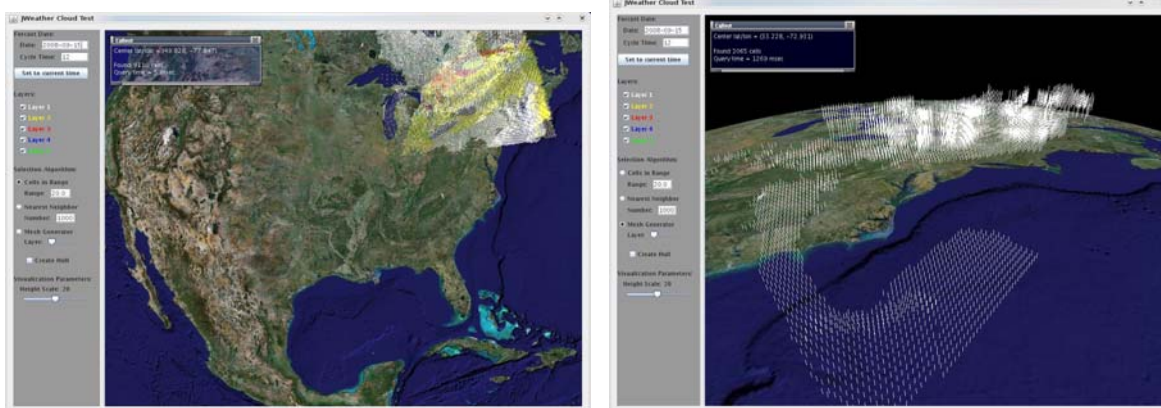


Figure 122: JWeather User Interface

4.6.5 Collaboration

We met with Prologic to get an overview of STORM, discuss weather related issues and demonstrate weather related software developed here. We provided them with a distribution (Figure 123) of the JWeather application.

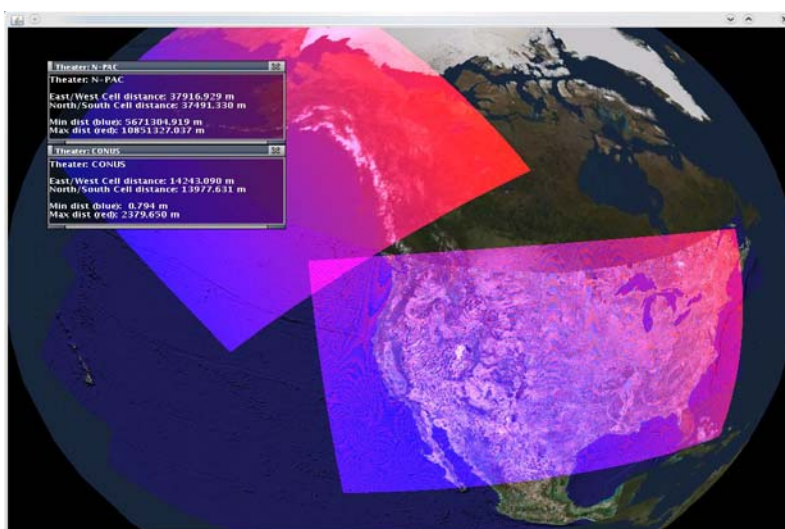


Figure 123: Screenshot of the JWeather Version Distributed to Prologic

4.7 Wacom Notepad

The WACOM tablet is a pressure-sensitive pen interactive display designed to support drawing and sketching in a more natural way than using a mouse. We were tasked to provide a Java interface to communicate with the WACOM device as well as to create a Java based

drawing application to experiment with advanced interaction techniques. This effort was started under the previous JView support contract, and enhanced under this one.

4.7.1 Drawing Curved Lines

CACI developed the NotePad application to interact with the WACOM device to draw and erase smoothly curved lines with different intensities. The problem with drawing lines or shapes using the Java2D API is that they can only have one thickness and one color per line. These problems could possibly be addressed using custom implementations of certain Java2D components, however we found a less cumbersome solution. We also explored many other avenues and found that the most successful method to draw curved lines on the display is to manually calculate and fill each individual pixel that fits a Bezier curve.

A Bezier curve is a curved line calculated by four control points. The first and last points are where the line starts and stops and the middle control points determine how the curve bends. During this period we found a better way to calculate the center control points for a Bezier curve. Given three points from the WACOM device, we find a line segment tangent to the curve at the center point, which is parallel to the line from the first control point to the last control point. The control point distances from the center point are scaled along the tangent line. The first one is a scaled distance based from the length from the start point to the center point. The second control point is positioned based on the length from the center point to the end point. Figure 124 (left) illustrates how the control points are calculated. On the right side of Figure 124 shows a zoomed in view of the line being drawn, where the yellow line is the straight line (based on the original points captured by the WACOM) and the green line is the tangent line showing the control points on either end.

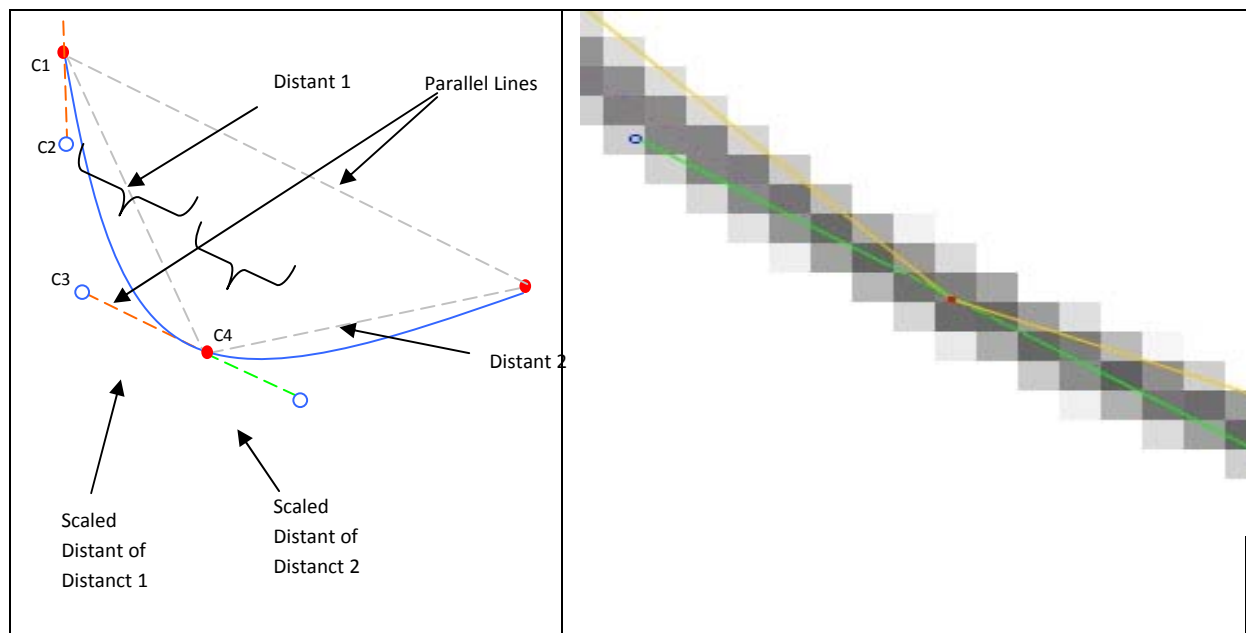


Figure 124: Bezier Control Points

4.7.2 Anti Aliasing

Once the Bezier curve has been defined and the curve points have been found, then the pixels to be filled are identified. We find candidate pixels by iterating through each of the curve points and calculating the maximum distance (in pixels) from the point that might need to be filled while corresponding to the width of the curve stroke. Each main point (points captured by the WACOM device) contains pressure information as well as a minimum and maximum width. With this information we can calculate the line stroke width at each main point, which corresponds to a circle centered about the point.

Pixels are then found for each circle starting with the first circle going to the last circle of the Bezier curve. If the distance from the center of the pixel to the center of the circle, minus the pixel offset is less than the radius of the circle (the stroke width) then the pixel is drawn and the intensity is calculated based on the distance. Once the pixels are found they are added to the list of pixels to be drawn. If a pixel is covered by more than one circle, the intensity is calculated as the maximum intensity value calculated from all of the circles. Figure 125 (top) shows an example of how circle overlaps are resolved, and Figure 125 (bottom) illustrates how pixel intensities are calculated.

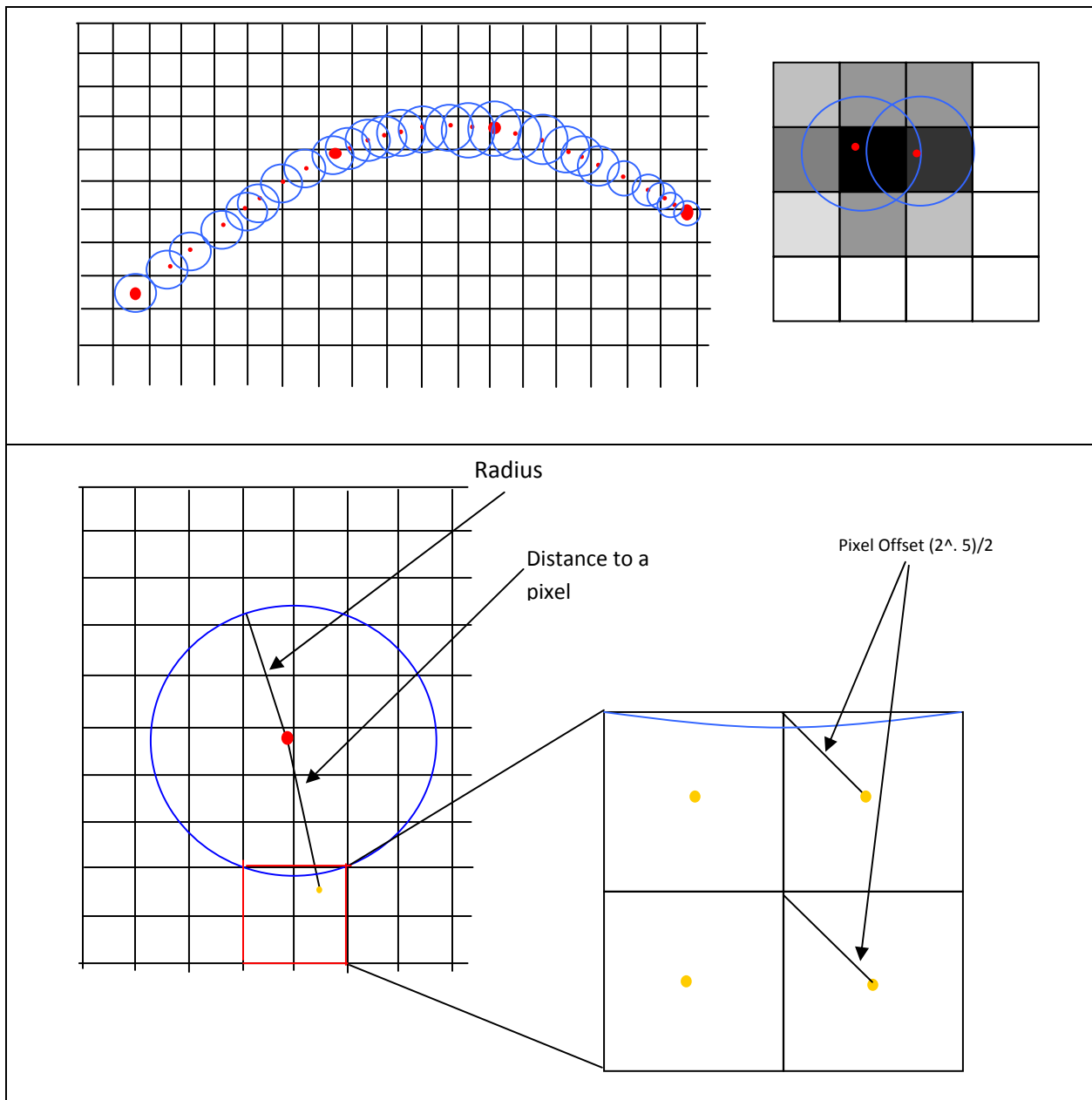


Figure 125: Pixel value under circle

4.7.3 Performance

The initial performance of the notepad application was not sufficient for interactive use. While the user was using the WACOM device, the lines that were being drawn lagged behind the location of the pen. Taking advantage of earlier research in this area, we implemented a tiled drawing system. The drawing area is divided into a grid of rectangular tiles. Each tile is

responsible for rendering itself to its assigned area on the screen. Tiles cache the image that they have most recently drawn, which avoids the need to recalculate the Bezier curves until a new curve is drawn on the tile, thus increasing the overall rendering performance.

Having each tile cache rendering results means that upon updating only the image needs to be drawn, the rasterization of the lines does not need to be re-computed. This makes the rendering process much faster for most cases. In Figure 126, the Tiled Notepad is showing lines drawn with a line width of 40 to 50 pixels, which is something that the old drawing system could not handle with a reasonable response time.

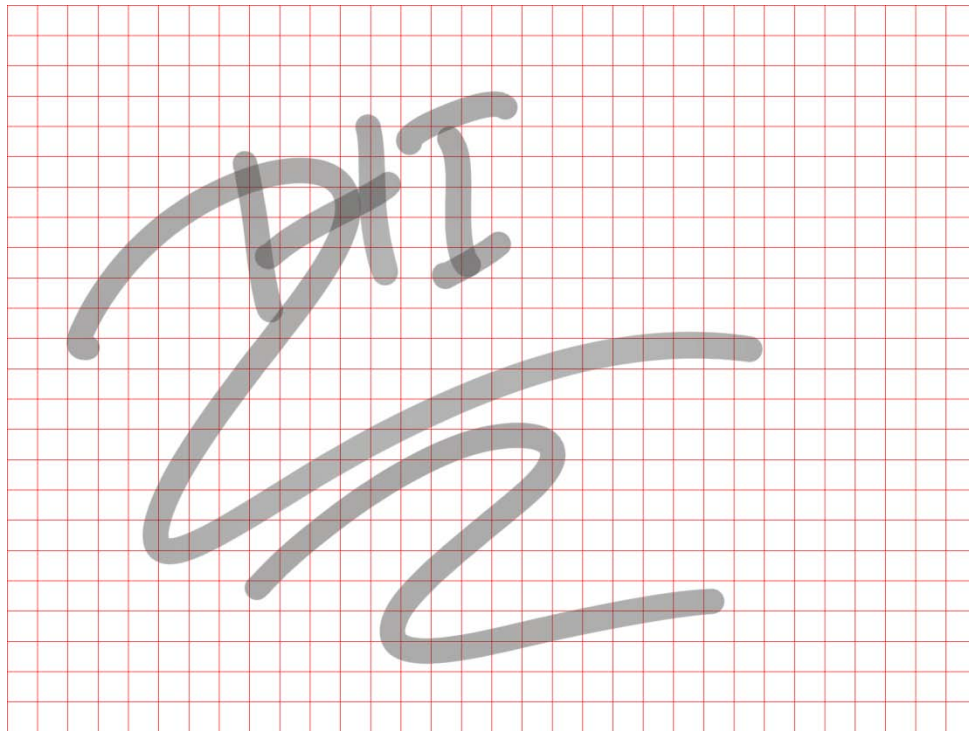


Figure 126: Tiled NotePad

4.7.4 Multiple Pages

We added support for multiple pages to the Notepad application so that users can switch pages using the keys on the WACOM tablet. This capability is vital for the experiment that the application was designed to support. We also added a reference number giving the users feedback on what page they are currently viewing. The reference number can either be drawn on the lower right corner of the page or pop up briefly while changing pages.

4.7.5 World “Drawer”

The World Drawer is a demonstration application to illustrate the concept and capability of hand drawing annotations. People today are annotating paper maps to incorporate information about a particular area. The annotations on the JView World element maps are then scanned back into a computer system for analysis. Using the WACOM tablet and the Notepad code we developed the World Drawer to display annotations on the World. The World Drawer has two modes for interaction: a navigate mode and a draw mode. The navigate mode is used for manipulating the viewing direction and location of the scene using the pen and WACOM tablet. The drawing mode is used to draw annotations on the World. Draw mode is activated by pressing the function key of the WACOM tablet. When this key is pressed, the World application is suspended and a screen shot is captured of the WACOM display. Then this image is used as a background for the drawing. The drawing is done with the same code used by the notepad application. After the user is finished drawing they click the same function key that started the drawing process and then the image is converted to a texture to be placed on the world. The conversion process takes the hand drawn points and converts the X/Y values to corresponding longitude/ latitude values. Converting the image to a geo-referenced image in this fashion allows the texture to be applied directly to the World, similar to how satellite imagery is applied. The image coordinates are mapped such that the top of the image corresponds to the highest latitude and the western portion of the image corresponds to the lowest (west-most) longitude. Figure 127 shows textures on the world that have been created using the world drawer.

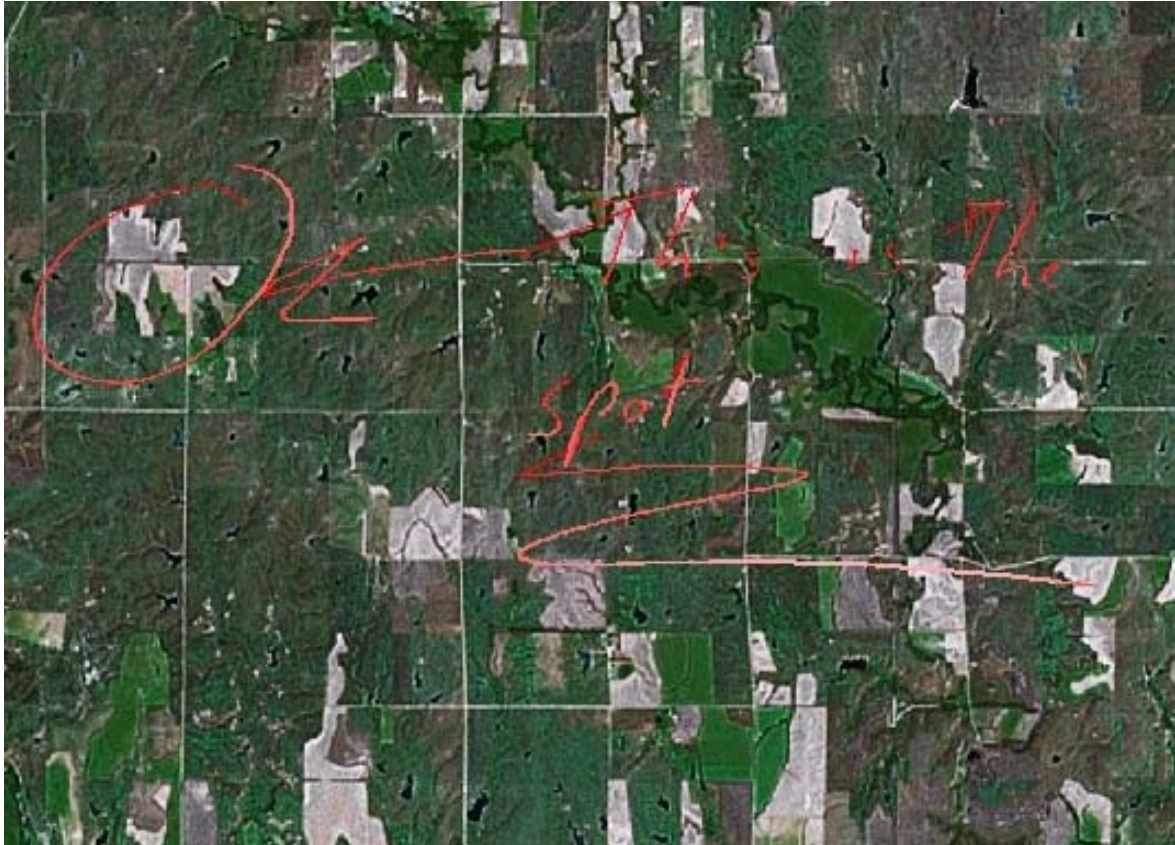


Figure 127: World Drawer

4.8 3D Model Tools

4.8.1 Model Preview Application

We developed a new JView based application to preview files in the various 3D model formats supported by JView. The application recursively scans a set of user-specified directories for model files. For any models found, the application generates a small preview image (if one has not been generated previously). After identifying models and generating thumbnails, the application presents a list of the available models, with the preview images embedded in the list (Figure 128). Users can select a model from the list to load it using JView's ElementViewer (Figure 129).

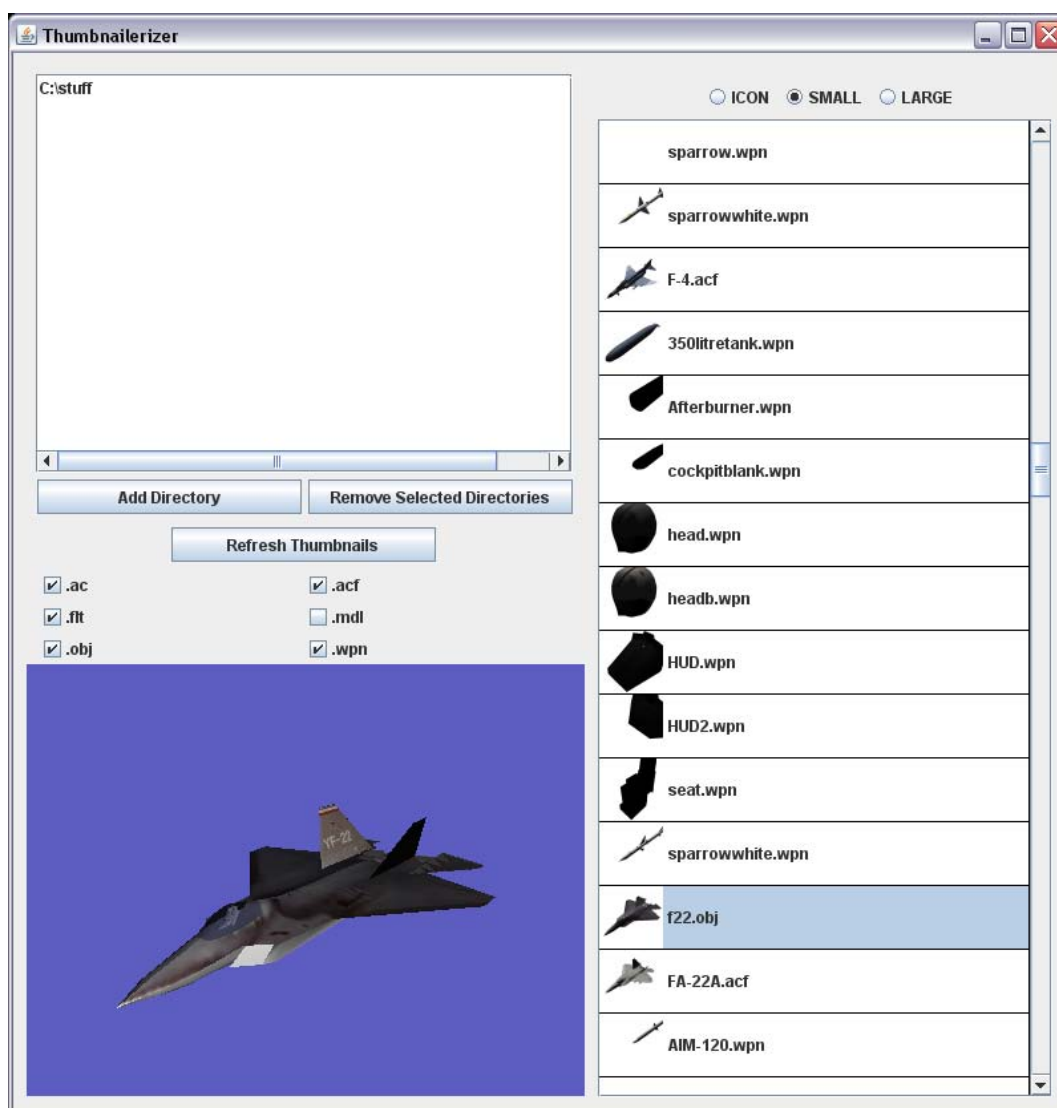


Figure 128: Model Preview Application

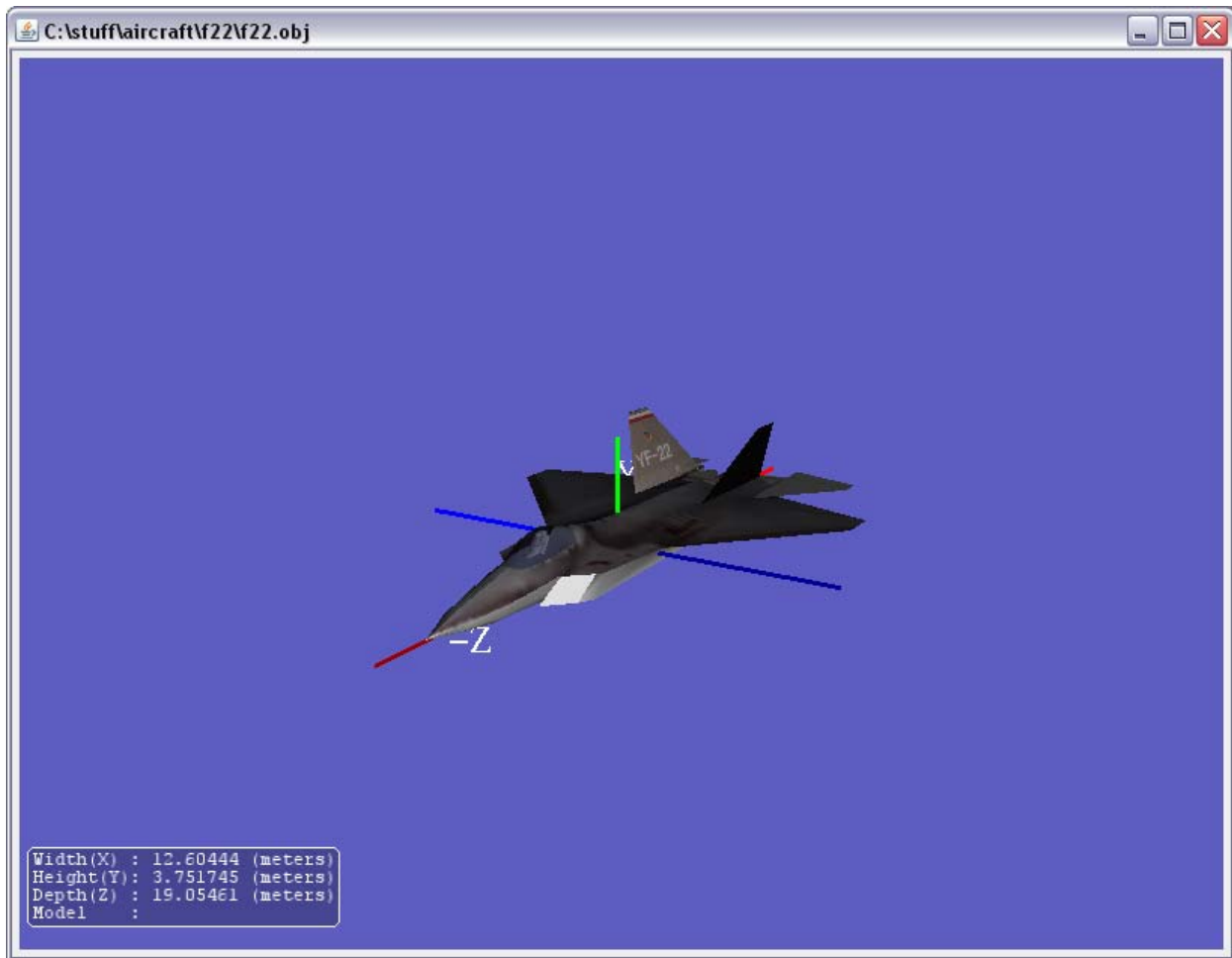


Figure 129: A model loaded in the ElementViewer, launched from the model previewer application

Many of the utilities that were developed for this application have been incorporated for general use in JView. These include tools to find the spatial bounds of a model, to automatically position the camera so the model is completely in view, and to automatically generate thumbnail images from one or more model files.

4.8.2 FLT Model Adjustment Tool

The FLT Model degree of freedom (DOF) adjustment tool is an application that we wrote to fix and save degree of freedom nodes in FLT models. Currently FLT models that are created using the PolyTrans tool do not articulate correctly because there are DOF nodes present that contain invalid data. Degree of freedom nodes define an axis of rotation using three points (origin, point1, and point2). The origin is the center point of the axis of rotation. Point1 defines a point that lies on the X axis, so this point defines where the X axis will be pointing. Point2

defines a point on the XY plane and with this point, point1 and the origin a plane is defined. Figure 130 shows the application where users can change parameters like origin, points, or ranges.

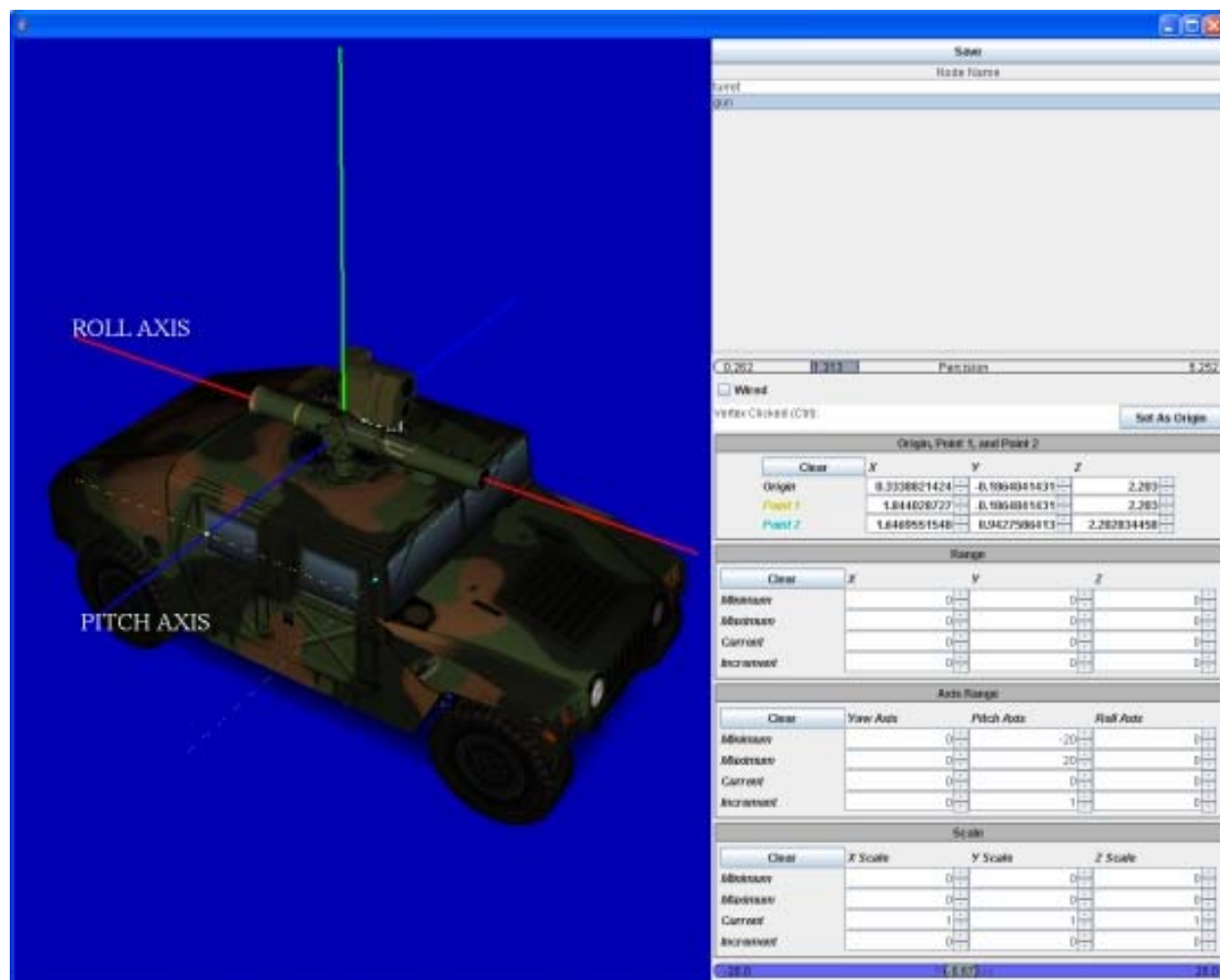


Figure 130: DOF fixer

Users can change DOF nodes of an FLT model using the drag and drop feature provided by the Element Viewer. Once a model is loaded the user can pick the DoF node they want to change in the list on the upper right corner. In Figure 131 the origin, point1 and point2 have been moved show the effect of a pitch rotation. In it we can see the gun rotate around the pitch axis when changing the pitch on the bottom right of Figure 130.

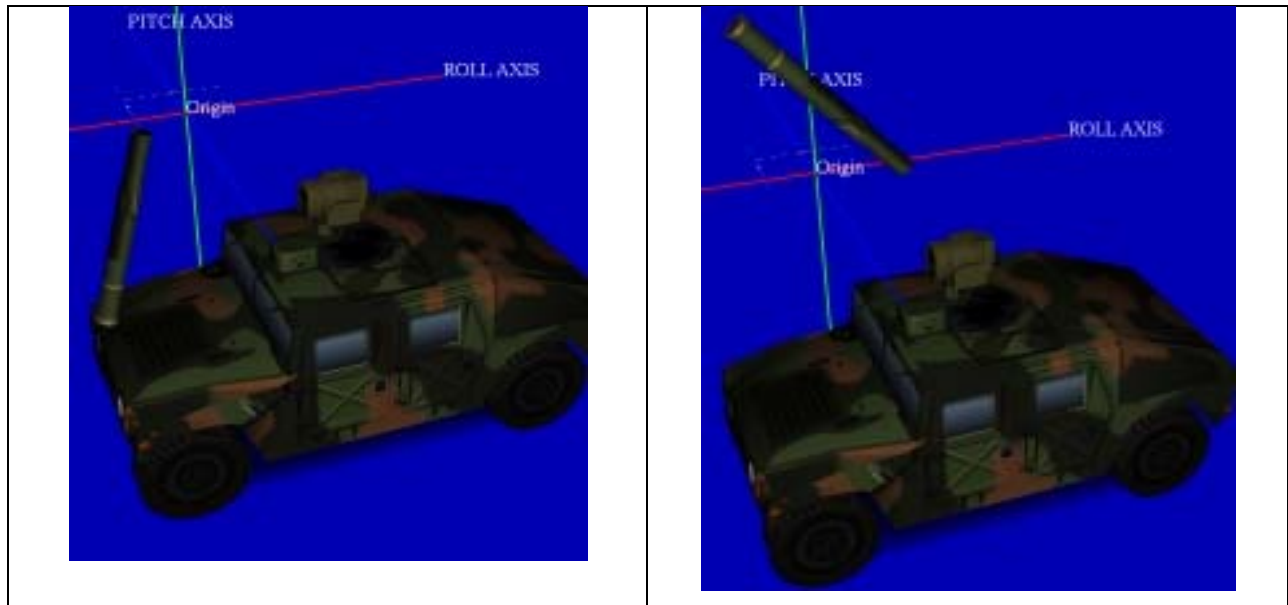


Figure 131: DOF Axis of Rotation

4.9 JView Demo Browser

The JView Demo Browser application provides a number of simple demonstration applications that illustrate how to use various utilities and subsystems in JView. Each demonstration is accompanied by a description of what it does, and well commented source code showing how it works. This is the best form of documentation for JView users since it provides precise (source code) instructions for each feature.

Over the course of this effort we incorporated over 100 new demonstrations to showcase newly available functionality, and maintained the existing set of demonstrations as JView APIs were changed. The new demonstrations range from very simple (e.g. adding textual labels to a 3D scene) to highly complex (simulating atmospheric light scattering in a GLSL shader program). Figure 132 shows screenshots of a small sampling of the demonstrations we developed.

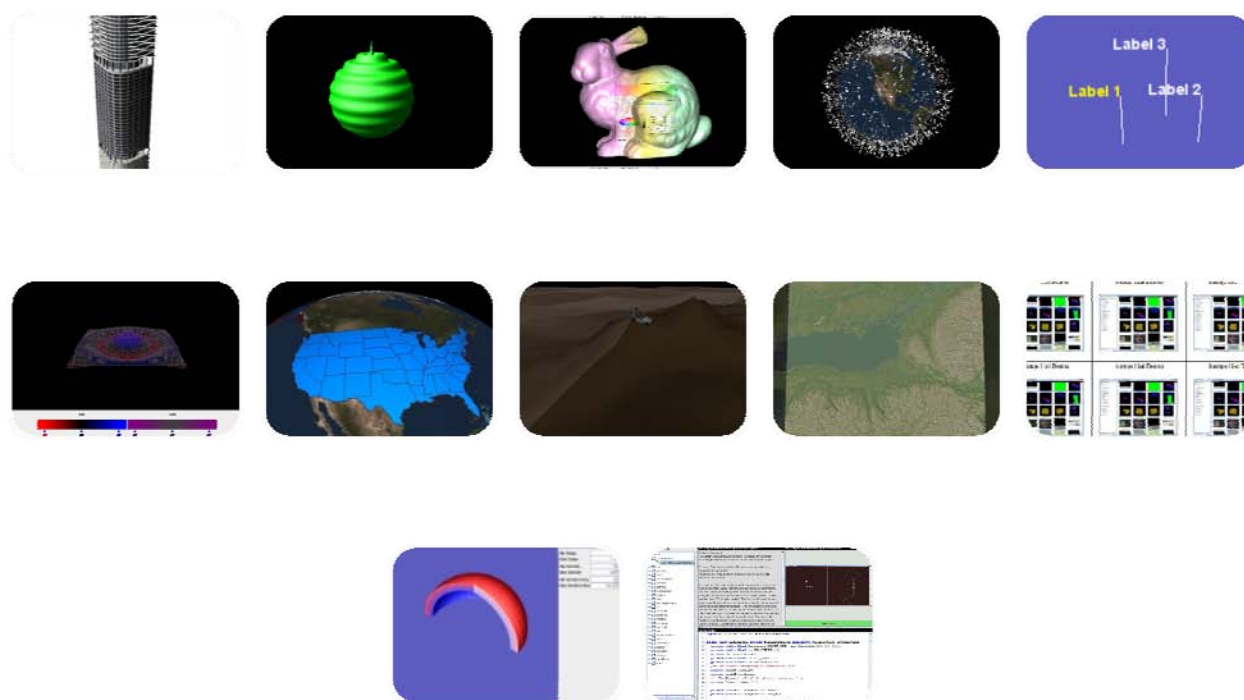


Figure 132: Images from Several of the Demo Browser Demonstration Applications

4.10 3D Models

Many JView based applications use digital assets (3D models) to represent various real-world objects in a visualization. In order to circumvent licensing and copyright issues that may impact our ability to distribute these assets with JView based applications, we developed a number of government owned models. In addition to avoiding issues with model distribution, we were able to respond directly and quickly to the specific needs of each application. For example, the CUNE Viewer application presents a visualization of captured data from real-world flight tests of a UAV. We were able to produce a model of the exact UAV being used for these tests, improving the level of realism in the visualization, as well as providing an accurate picture for analysis.

Throughout the course of this effort, we developed dozens of models of various Air Force, Cyber, and Ground Based assets that can be used and redistributed by any government agency. Several examples are shown in Figure 133.



Figure 133: Several Models Developed by CACI Rendered with JView.

5.0 CONCLUSIONS

This effort yielded significant contributions to the information visualization capabilities of the Air Force Research Laboratory. Perhaps the most important result of this effort is the enhancement of the JView toolkit. JView was originally designed to consolidate information visualization techniques into a library of developer tools that facilitate rapid development of domain specific applications. We have simultaneously improved JView's usability, added new visualization capabilities, and maintained currency with the latest graphics hardware capabilities as they advance.

A major part of the JView enhancements involved continued development of JView World and associated tools to support the large community of users with geographic information visualization needs. JView World offers a unique blend of flexibility, performance, and capability compared to other terrain visualization tools such as Google® Earth or NASA WorldWind. It has found its way into most of the applications being developed with JView, and the Terra Firma application helps to demonstrate its capabilities with an interface that feels familiar to users of other terrain visualization tools.

Another important aspect of this work was the development of new domain specific visualization tools. Applications such as the CUNE Viewer and JWeather are designed to assist researchers in their analysis and presentation of results as well as tools designed to enhance situational awareness and command and control capabilities in an operational setting. The Audit Trail Viewer from which JView was originally derived has also been maintained and enhanced with new capabilities.

The ACES Viewer, while originally designed to support NASA's efforts to increase the capacity of the NAS, has become a platform for highly customizable visualization tools, and has become the foundation for other work such as the User Defined Operational Picture, and a new weather visualization effort. This tool effectively reduces the development effort on projects that use it by providing reusable infrastructure and visualization mechanisms that leverage JView and other libraries.

Finally, we have incorporated recent work from the information visualization community into JView, exposing sophisticated visualization capabilities such as text decluttering, and graph layout in an approachable manner. All of this work ultimately benefits the Air Force in the short term by providing visualization capabilities for specific projects and in the long term by reducing the effort needed to develop custom visualization tools for any program.

REFERENCES

- Agrawala, M., Heer, J., "Software Design Patterns for Information Visualization," *IEEE Transactions on Visualization and Computer Graphics*, Vol. 12, No. 5, 2006.
- Beyer, K., Chen, G., Donjerkovic, D., Lawande, S., Livny, M., Myllymaki, J., Ramakrishnan, R., Wenger, K., "DEVise: Integrated Querying and Visual Exploration of Large Datasets," 1997.
- Bosch, R., Gerth, J., Hanrahan, P., Rsenblum, M., Stolte, C., Tang, D., "Rivet: A Flexible Environment for Computer Systems Visualization," *Computer Graphics*, Feb 2000.
- Chen, W., Pfister, H., Ren, L., Zwicker, M., "Hardware-Accelerated Adaptive EWA Volume Splatting," 2004.
- Chuah, M.C., Kerpedjiev, S., Kolojejechick, J., Lucas, P., Roth, S.F., "Towards and Information Visualization Workspace: Combining Multiple Means of Expression," *Human-Computer Interaction Journal*, Vol. 12, No. 1 & 2, pp. 131 – 185, 1997.
- Codd, E.F., "A Relational Model of Data for Large Shared Data Banks," *Information Retrieval, Communication of the ACM*, Vol. 13, No. 6, pp. 377 – 387, Jun 1970.
- Davidson, R., David, H., "Drawing Graphs Nicely Using Simulated Annealing," *ACM Transactions on Graphics*, Vol. 15, No. 4, pp. 301-330, Oct. 1996.
- Guan, X., Hong, W., Kaufman, A., McDonnell, K.T., Mueller, K. Neophytou, N., Qin, H., "GPU-Accelerated Volume Splatting with Elliptical RBFs," *Eurographics/IEEE-VGTC Symposium on Visualization*, 2006.
- Jankun-Kelly, T.J., "Visualizting Visualization: A Model and Framework for Visualization Exploration," University of California, 2003.
- Kerthick, M., Kolojejechick, J., Roth, S.F., "An Interactive Visual Query Environment for Exploring Data," *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST '97)*, pp 189 – 198, Oct 1997.
- Lacroute, P.G., *Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation*, CSL-TR-95-678, Stanford University, Stanford, CA, Sep 1995.
- McVay, A., Krisher, D., Fisher, P., *JView Visualization for Virtual Airspace Modeling and Simulation*, AFRL-RI-RS-TR-2009-79, CACI Technologies Inc., Rome, NY, 13441, Apr 2009.
- Westover, L., "Footprint Evaluation for Volume Rendering," *Computer Graphics*, Vol. 24, No. 4, pp. 367 - 376, Aug 1990.

ACRONYMS

2D	Two-Dimensional
3D	Three Dimensional
4D	Four-Dimensional
ACES	Airspace Concepts Evaluation System
AFRL	Air Force Research Laboratory
AFRL/RISB	Air Force Research Laboratory's Decision Support Systems Branch
AFWA	Air Force Weather Agency
AGL	Above Ground Level
AGP	Advanced Graphics Processor
APATS	Antenna Patterns Analysis Tool Set
API	Application Programming Interface
ATV	Audit Trail Viewer
BFS	Breadth First
CACI	Consolidated Analysis Centers, Inc
CLOD	Continuous Level-Of-Detail
CPU	Central Processing Unit
CSV	Comma Separated Value(s)
CTAS	Center Tracon Automation System
CUNE	Characteristic of UAV Network Environments
DCF	Diagnostic Cloud Forecast Model
DoF	Degree of Freedom
DRAM	Dynamic Random Access Memory
DTED	Digital Terrain Elevation Data
FAA	Federal Aviation Administration
GB	Gigabyte
GeoTIFF	Geostationary Earth Orbit Tagged Image File Format
GIS	Geographic Information System
GL	Graphics Language
GLSL	OpenGL Shading Language
GLU	Graphics Language Utility
GPS	Global Positioning System
GRIB	Gridded Binary
GUI	Graphics User Interface
HUD	Heads Up Display
IDE	Integrated Development Environment
IV4D	Interactive Visualization in 4D
JDBC	database connection standard used by Java programs

JMBL	Joint METOC Broker Language
JOGL	Java Open GL (Graphics Language)
JWIS	Joint Weather Impact System
Kd	K-dimensional
KML	Keyhole Markup Language
KMZ	Keyhole Markup Language, Zipped
LAN	Local Area Network
LoD	Level Of Detail
METOC	Meteorological and Oceanographic
MSL	Mean Sea Level
NAS	National Airspace System
NASA	National Aeronautics and Space Administration
NM	New Mexico
OSGi	Open Services Gateway Initiative
OSX	Macintosh Operating System X
RPF	Raster Product Format
SAX	Simple API for XML
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SSL	Secure Socket Layer
TIFF	Tagged Image File Format
UAV	Unmanned Aerial Vehicle
URL	Uniform Resource Locator
VBO	Vertex Buffer Objects
VM	Virtual Machine
VRAM	Video Random Access Memory
WATP	World Automated Test Program
WGS84	World Geodetic System 1984
WDA	Weather Decision Aid
WMS	Web Map Service
XML	Extensible Markup Language